Mastering Tcl/Tk

From Beginner to GUI Developer

All code tested with Tcl/Tk 8.6.17

Based on official Tcl/Tk 8.6 documentation from tcl-lang.org

Table of Contents

Chapter 1. Introduction to Tcl and Tk
Chapter 2. The Tcl Syntax Model
Chapter 3. Variables, Quoting, and Scope
Chapter 4. Expressions and Control Flow
Chapter 5. Procedures and Scope
Chapter 6. Strings, Lists, and Dictionaries
Chapter 7. File I/O, Errors, and Packages
Chapter 8. Regular Expressions
Chapter 9. Tk Fundamentals
Chapter 10. Widgets
Chapter 11. Geometry Managers
Chapter 12. Events and Bindings
Chapter 13. Canvas Graphics
Chapter 14. Menus, Dialogs, and State
Chapter 15. Project: Calculator
Chapter 16. Project: Notepad
Chapter 17. Project: Drawing App
Chapter 18. Project: Tic-Tac-Toe Game

A hands-on tutorial built on official Tcl/Tk 8.6 documentation

This book teaches the Tcl scripting language and the Tk GUI toolkit from the ground up. Every code example in this book has been tested and is drawn from the companion script set that accompanies each chapter. The book follows the official Tcl/Tk 8.6 manual pages hosted at tcl-lang.org, which are cited throughout so you can dig deeper into any command.

Tcl (Tool Command Language) is a string-based, dynamically typed scripting language created by John Ousterhout in 1988. Tk is its companion toolkit for building graphical user interfaces. Together they form a remarkably compact yet powerful platform for everything from rapid prototyping and test automation to full-featured desktop applications. Tcl's design centers on a single, simple rule set: everything is a command, and substitution happens in a predictable, left-to-right order. Master that rule set and you have mastered the heart of the language (Tcl manual page).

How to use this book

The book has 18 chapters. Chapters 1-8 cover core Tcl: syntax, variables, expressions, control flow, procedures, strings, lists, dictionaries, file I/O, and regular expressions. Chapters 9-14 cover Tk: windows, widgets, layout, events, the canvas, and menus/dialogs. Chapters 15-18 are hands-on projects that combine everything into complete applications: a calculator, a notepad, a drawing program, and a tic-tac-toe game.

Each chapter follows the same structure:

1. A short introduction.
2. Concept explanations with references to the official man pages.
3. The *complete* tested script for that chapter, presented in a code block, followed by a line-by-line walkthrough.
4. A "What You Learned" summary.
5. Two or three exercises.
6. Citations to the relevant official documentation URLs.

> **Note (callout boxes):** Throughout the book you will see three kinds of
> callout boxes. **Tip** boxes highlight a useful technique. **Warning** boxes
> flag a common pitfall. **Note** boxes add context or background.

Running the examples

Console (chapters 1-8) examples run with the `tclsh` interpreter:

```
tclsh ch01_hello.tcl
```

GUI (chapters 9-18) examples run with the `wish` shell, which loads Tk automatically:

```
wish ch09_first_gui.tcl
```

The scripts assume Tcl/Tk 8.6 or later. See the official downloads at tcl-lang.org.

Chapter 1: Introduction to Tcl and Tk

Tcl is a lightweight, embeddable scripting language with a famously simple syntax: a script is a sequence of commands, each command is a sequence of words, and the first word names the command to run. This chapter introduces the language, its history, how to install it, and how to run your first program.

1.1 What is Tcl?

Tcl (Tool Command Language) was designed by John Ousterhout at the University of California, Berkeley, beginning in 1988. Its goals were to be a reusable, embeddable scripting core that applications could extend with their own commands, and to have an uncluttered syntax that is easy to learn. Tk, the GUI toolkit, followed shortly after and became one of the most popular reasons to use Tcl. The language is string-based and dynamically typed: every value is a string (even if, internally, Tcl caches typed representations for speed), and variables are created on first assignment.

The official Tcl manual page describes the foundation: a Tcl script is a string containing one or more commands separated by newlines or semicolons, and each command is a series of words separated by whitespace, where the first word is the command name and the rest are arguments (Tcl manual page).

1.2 Installing Tcl/Tk

Tcl/Tk 8.6 is available pre-built for Windows, macOS, and Linux from the official site at tcl-lang.org. On most Linux distributions you can install the `tcl` and `tk` packages. On macOS, the ActiveTcl distribution or Homebrew's `tcl-tk` are common choices. After installation you will have two

interpreters:

- `tclsh` - the shell for running console scripts.
- `wish` - the windowing shell, which also loads Tk for GUI programs.

Verify your installation at a terminal:

```
tclsh8.6
% puts $tcl_version
8.6
% exit
```

1.3 Your first program

The traditional first program prints a greeting. In Tcl, printing to standard output is done with the `puts` command. The `set` command creates a variable and assigns it a value, and the `$` sign performs variable substitution. The `expr` command evaluates an expression, and square brackets `[...]` perform command substitution, inserting the result of the inner command into the outer one. The official reference for `set` and `puts` lives on the `set` man page (set manual page),
and arithmetic is documented on the `expr` man page
(expr manual page).

Here is the complete tested script for this chapter, `ch01_hello.tcl`:

```
# ch01_hello.tcl — First Tcl program
puts "Hello, World!"

# Variables and substitution
set name "Tcl"
set version 8.6
puts "Welcome to $name $version"

# Command substitution
set result [expr {3 + 4}]
puts "3 + 4 = $result"
```

1.4 Line-by-line walkthrough

- `puts "Hello, World!"` - The `puts` command writes its argument string to standard output, followed by a newline. `"Hello, World!"` is a single word grouped by double quotes, so the spaces are preserved as part of the argument.
- `set name "Tcl"` - `set` with two arguments assigns the value `"Tcl"` to the variable `name` and returns the value. Tcl creates the variable automatically; there is no separate declaration step.
- `set version 8.6` - assigns the value `8.6` (a string) to `version`. Notice that 8.6 is not declared as a number; it is stored as the string `"8.6"`.

- `puts "Welcome to $name $version"` - The double-quoted string allows **variable substitution**: `$name` is replaced with the value of `name`, and `$version` with the value of `version`, before `puts` runs.
- `set result [expr {3 + 4}]` - This line shows two things at once. The square brackets perform **command substitution**: the `expr {3 + 4}` command is evaluated first, and its result (7) replaces the bracketed text. That result is then assigned to `result` by `set`. The braces `{ }` around the expression tell `expr` to treat the contents as a single argument; this is the recommended, efficient form.
- `puts "3 + 4 = $result"` - substitutes `$result` into the output string.

> **Tip:** Always brace your `expr` arguments (write `expr {$a + $b}`, not
> `expr $a + $b`). Bracing lets the bytecode compiler evaluate the expression
> efficiently and prevents double substitution surprises.

1.5 The two interpreters

`tclsh` is a plain shell for scripts that do not need a GUI. It reads a script from a file (or from standard input in interactive mode) and runs it to completion, then exits. `wish` does the same but also creates a root window (`.`) and starts the Tk event loop, so it is the interpreter you use for any program that displays a window. Chapters 1-8 of this book run under `tclsh`; chapters 9-18 run under `wish`.

Concepts in Depth

Why "everything is a string" matters

Tcl's single most distinctive design choice is that every value is, at the logical level, a string. When you write `set version 8.6`, the variable holds the string `"8.6"`. When a command like `expr` needs a number, it parses the string on the fly; when a command like `string length` needs text, it already has text. There are no separate integer, float, or boolean types to declare, and no type-casting syntax to learn. This uniformity is what makes Tcl's twelve-rule syntax model sufficient to describe the entire language: because commands always receive strings and always return strings, substitution (`$`, `[...]`, `\`) is the only glue needed between them.

Internally, Tcl is not naive about performance. The interpreter caches an efficient internal representation (an integer, a double, a list, a byte array) for a value the first time a command needs it, and reuses that representation as long as the string form does not change. This is called *shimmering* when a

value is repeatedly converted back and forth between representations. You rarely need to think about it, but it explains two practical rules of thumb: brace your `expr` arguments (so the expression is compiled to bytecode once), and avoid bouncing the same value through commands that expect incompatible representations in a tight loop.

The read-eval-print loop and scripts

A Tcl script is simply a string. The interpreter parses it into commands, substitutes and evaluates each one in turn, and the result of the last command becomes the result of the script. This is equally true for a file fed to `tclsh`, for a string passed to `eval`, and for the body of a `proc`. Because a script is a string, you can build, store, and run code dynamically - a property that makes Tcl exceptionally good at metaprogramming, configuration, and test scripting, but also one that demands care when handling untrusted input (which is why bracing `expr` and avoiding `eval` on raw data are both security measures, not just style choices).

Embedding and extension

Tcl was designed from day one to be embedded inside larger applications and extended with new commands written in C (or other languages). The `package` mechanism you will meet in Chapter 7 is the user-facing side of that philosophy: a package is a versioned bundle of commands that any script can load with `package require`. Tk itself is just such a package, which is why `package require Tk` is the bridge between the console chapters and the GUI chapters of this book.

A note on versions

This book targets Tcl/Tk 8.6, the long-standing stable release. Features such as `try`/`trap`, `lmap`, `dict`, and the `apply` command all date from 8.5 or 8.6 and are used throughout. Tcl 8.7 and 9.0 add features like named capture groups in regular expressions and additional string operations, but the 8.6 idioms shown here remain correct and portable.

What You Learned

- Tcl is a string-based scripting language; every value is conceptually a string.
- A script is a sequence of commands; a command is a sequence of words; the first word is the command name.
- `puts` prints to standard output; `set` creates and assigns variables.
- `$` performs variable substitution and `[...]` performs command

substitution.
- `expr` evaluates expressions; brace its argument for efficiency and safety.
- `tclsh` runs console scripts; `wish` runs GUI scripts and starts the Tk event loop.

Exercises

1. Modify `ch01_hello.tcl` so it asks for your name using the `gets stdin` command and greets you by name.
2. Add two more variables and print their sum using `expr`, formatting the output as `"The sum of X and Y is Z"`.
3. Use `set` with a single argument (e.g. `set name`) to read a variable's value, and print the result. What happens if the variable does not exist?

Chapter 2: The Tcl Syntax Model

This chapter unpacks the rules that govern how Tcl parses every line of every script. Tcl has no reserved words and almost no special syntax outside of its twelve substitution and grouping rules, so understanding these rules once means you can read any Tcl code.

2.1 Commands, words, and arguments

According to the official manual page, a Tcl script is a string containing one or more commands, and each command is a series of words separated by whitespace. The first word is the command name; the remaining words are the arguments. Commands are normally terminated by a newline or a semicolon, unless that terminator is quoted, enclosed in braces, or escaped with a backslash (Tcl manual page).

2.2 The three kinds of substitution

Tcl performs exactly three forms of substitution, all from left to right, each completed before the next begins:

1. **Variable substitution** with `$`, e.g. `$count`.
2. **Command substitution** with `[...]`.
3. **Backslash substitution** with sequences like `\t`, `\n`, `\u20ac`.

2.3 Grouping: braces vs. quotes

- **Braces** `{ }` group text into a single word and *suppress* substitution.
This is why `expr {$a + $b}` is safe: the braces protect the expression from being substituted before `expr` sees it.
- **Double quotes** `" "` group text into a single word but *allow* substitution, so variables and command substitutions inside quotes are expanded.

2.4 The tested script

`ch02_syntax.tcl` demonstrates each of these mechanisms:

```
# ch02_syntax.tcl — Tcl syntax model: commands, words, substitution

# Command structure: command arg1 arg2 ...
puts "Words are separated by spaces"

# Variable substitution with $
set fruit "apple"
set count 5
puts "I have $count ${fruit}s"

# Command substitution with []
set length [string length $fruit]
puts "Length of '$fruit' is $length"

# Backslash substitution
puts "Tab\there"
puts "Newline\nhere"
puts "Unicode: \u20ac"

# Brace grouping: no substitution
puts {Braces prevent $substitution}

# Quote grouping: substitution happens
puts "Quotes allow $count substitution"

# Semicolons separate commands
set a 1; set b 2; puts "a=$a b=$b"

# Comments
# This is a full-line comment
puts "ok" ;# This is an inline comment
```

2.5 Line-by-line walkthrough

- `puts "Words are separated by spaces"` - the quoted string is one word passed to `puts`.
- `set fruit "apple"` / `set count 5` - create two variables.
- `puts "I have $count ${fruit}s"` - shows two variable-substitution forms. `$count` substitutes directly. `${fruit}s` uses the brace form `${name}` to disambiguate the variable name from the trailing `s`; without braces, `$fruits` would look for a variable named `fruits`.
- `set length [string length $fruit]` - command substitution: the `string`

command runs, returns 5, and that value is assigned to `length`.
- `puts "Length of '$fruit' is $length"` - both variables substituted inside quotes.
- `puts "Tab\there"` / `puts "Newline\nhere"` / `puts "Unicode: \u20ac"` - backslash substitution produces a tab, a newline, and the Euro sign (code point U+20AC).
- `puts {Braces prevent $substitution}` - braces suppress substitution, so the literal text `$substitution` is printed.
- `puts "Quotes allow $count substitution"` - quotes allow it, so 5 is printed.
- `set a 1; set b 2; puts "a=$a b=$b"` - semicolons separate three commands on one line.
- `# This is a full-line comment` - a # at the start of a command position begins a comment.
- `puts "ok" ;# This is an inline comment` - after a command, `;#` starts an inline comment. Note that # only acts as a comment where a command would be expected; it is a normal character inside a word.

> **Warning:** Comments in Tcl are a subtlety of the parser, not a separate
> language feature. A # only starts a comment where the parser expects the
> first word of a command. Inside braces or quotes, # is just text. Also, a
> # comment still has to be syntactically valid Tcl (balanced braces), or
> you will get confusing errors.

2.6 The twelve rules, summarized

The manual page presents the syntax as twelve rules. In short: scripts are sequences of commands (rule 1); commands are sequences of words (2); words are separated by whitespace (3); the first word is the command name (4); the rest are arguments (5); a command ends at a newline or semicolon unless quoted, braced, or escaped (6); braces group a word (7); quotes group a word (8); braces suppress substitution except backslash-newline (9); quotes allow substitution (10); there are three substitution types (11); and substitutions happen left to right, each fully completed first (12) (Tcl manual page).

Concepts in Depth

Substitution is a single pass, left to right

A common source of confusion is the belief that substitution is recursive or

that it happens "until nothing changes." It does not. Tcl makes exactly one left-to-right pass over each word, performing variable, command, and backslash substitution as it goes, and the result is handed to the command. If a substitution *produces* text that looks like another substitution (for example, a variable whose value is the string `"$x"`), that produced text is not re-substituted. This single-pass rule is what makes Tcl predictable and safe: you always know precisely what will be evaluated.

Why braces are the safe default

Because braces suppress substitution, anything inside braces reaches a command verbatim. This is essential for `expr` (so the expression is compiled, not pre-substituted), for `regexp` patterns (so backslashes reach the engine), for `proc` bodies, and for any literal that contains `$`, `[`, or `\`. A good mental model: use double quotes only when you specifically want substitution, and use braces everywhere else.

Continuation lines and formatting

A backslash immediately followed by a newline is removed (along with leading whitespace on the next line), so a long command can be split across physical lines for readability. This is different from a quoted newline (`\n`), which produces an actual newline character in the value. Confusing the two is a frequent beginner mistake.

Comments and the parser

A # begins a comment only where the parser expects the first word of a command. This means # inside a braced block passed to `proc` or `if` is treated as a comment *within that script* (because the block is itself a script), but a # in the middle of a word is just a character. It also means a comment line must still be syntactically valid Tcl - an unbalanced brace in a comment will cause an error, because the parser scans the whole line for word boundaries before discarding it as a comment.

Word separation and lists

Because words are separated by whitespace, a string with spaces is not the same as a single word unless it is grouped. This is the bridge between the syntax model and lists: a list is just a sequence of words, and the same quoting rules (braces and quotes) apply. Understanding Chapter 2 therefore prepares you for the list and `dict` commands in Chapter 6, which rely on identical quoting conventions.

What You Learned

- Commands are words separated by whitespace; the first word names the command.
- Newlines and semicolons terminate commands unless quoted, braced, or escaped.
- Three substitutions exist: variable (`$`), command (`[...]`), and backslash (`\...`).
- Braces suppress substitution; quotes allow it.
- `${name}` disambiguates variable names adjacent to other characters.
- `#` starts a comment only where a command is expected.

Exercises

1. Predict the output of `puts {[$x]}` when `x` is set to `hello`, and explain why no substitution occurs.
2. Rewrite the inline-comment line so that the comment uses braces instead of `;#`, and observe the difference.
3. Use backslash-newline to split a single `puts` command across two physical lines.

Chapter 3: Variables, Quoting, and Scope

Tcl variables hold strings and are created the moment you assign to them. This chapter covers scalar variables, array variables, reading variables with the single-argument `set`, removing them with `unset`, and multi-line strings. The central command is `set`, documented on the official man page (set manual page).

3.1 Scalar variables

A scalar variable holds a single value. You create one with `set varname value`, which assigns value to varname` and returns the value. You read a variable either with `$varname` substitution or with `set varname` (the single-argument form), which returns the current value.

3.2 Array variables

An array is a collection of named elements accessed with the syntax `arrName(element)`. Each element is itself a scalar string. Arrays are unordered and are not the same as lists or dictionaries. The `array` command provides operations such as `array names` (which returns the element names) and `array get` (which returns a flat list of name/value pairs). You cannot pass an entire array to a procedure by value; instead you use `upvar` (covered in Chapter 5).

3.3 The tested script

`ch03_variables.tcl` exercises scalars, arrays, iteration, string operations, the single-argument `set`, `unset`, and multi-line braced strings:

```
# ch03_variables.tcl — Variables, quoting, and scope

# Scalar variables
set x 42
set greeting "Hello"
puts "$greeting, x is $x"

# Array variables
set person(name) "Alice"
set person(age) 30
set person(city) "NYC"
puts "Name: $person(name), Age: $person(age)"
puts "Array names: [array names person]"

# Array iteration
foreach key [array names person] {
    puts "  $key = $person($key)"
}

# String operations on variables
set str "Hello World"
puts "Uppercase: [string toupper $str]"
puts "Length: [string length $str]"
puts "First word: [string range $str 0 4]"

# The set command with one arg reads a variable
set y [set x]
puts "y = $y (copied from x)"

# unset removes variables
unset y
# puts $y  ;# This would cause an error

# Multi-line strings with braces
set poem {
Roses are red,
Violets are blue,
Tcl is neat,
And scripts are too.
}
puts $poem
```

3.4 Line-by-line walkthrough

- `set x 42` / `set greeting "Hello"` - create two scalar variables.

- `puts "$greeting, x is $x"` - both substituted inside quotes.
- `set person(name) "Alice"` and friends - assign elements of the array `person`. Each `set person(key) value` creates or updates one element.
- `puts "Name: $person(name), Age: $person(age)"` - array element substitution uses `$arr(element)`.
- `puts "Array names: [array names person]"` - `array names` returns a list of the element names; the result of that command is substituted in.
- `foreach key [array names person] { puts " $key = $person($key)" }` - iterates over the element names and prints each name/value pair. Because array elements have no guaranteed order, the iteration order is unspecified.
- The `string` operations (`string toupper`, `string length`, `string range`) are a first taste of Chapter 6's string command; here they show that any command result can be assigned with `set` or printed with `puts`.
- `set y [set x]` - the single-argument `set x` reads the value of `x` (returning 42), which is then assigned to `y`. This is the way to read a variable whose name is not a legal variable-substitution form.
- `unset y` - removes the variable `y`. After this, referencing `$y` would raise an error.
- The `poem` variable is assigned a multi-line string using braces. Braces group the entire block, including newlines, into one value; no substitution happens inside. `puts $poem` prints it verbatim.

> **Tip:** Use braces for any literal string that contains characters Tcl
> would otherwise interpret (`$`, `[`, `\`). Braces are the safest quoting
> mechanism because they suppress substitution entirely (except
> backslash-newline for line continuation).

> **Note:** Arrays are convenient but have limitations: you cannot pass a whole
> array as a value, and they have no defined ordering. For ordered, passable
> key/value collections, prefer `dict` (Chapter 6).

Concepts in Depth

The set command's dual role

`set` is overloaded: with two arguments it assigns; with one argument it reads. This is not a quirk but a deliberate design that keeps the language small. Because reading uses the same command as writing, you can read a variable whose name contains characters that would confuse `$` substitution (such as a name with spaces or unusual characters) by calling `set` with that name as a single argument. The `$` syntax is pure sugar over `set`; internally `$name` is

essentially `[set name]`.

Variables are not declared

Tcl creates a variable the moment you assign to it, and destroys it with `unset`. There is no declaration, no typing, and no scope annotation at the assignment site. This makes scripts concise, but it also means a typo in a variable name silently creates a new, empty variable rather than raising an error. The `info exists varName` command lets you test for existence before use, which is important when reading optional configuration or optional command-line arguments.

Arrays vs. dictionaries

Arrays and dictionaries both map keys to values, but they are fundamentally different kinds of objects. An array is a *variable*: it exists in a scope, its elements are scalars, and you cannot pass the whole array as a value to a procedure (you must use `upvar` or `array get`/`array set`). A dictionary is a *value*: it is a first-class string (an even-length list of alternating keys and values) that you can store, pass, return, and nest. For new code, dictionaries are almost always the better choice; arrays remain useful when you need a mutable, scope-local lookup table that many procedures share via `upvar`.

Quoting and security

The same rules that govern grouping also govern safety. When you build a command from parts (for instance, a callback that includes a variable), use the `list` command to construct it so that special characters in the data do not break out of the word boundary. This principle returns in Chapter 15, where the calculator builds per-button commands with `[list digitPressed $label_text]` rather than string interpolation.

Linkage and traces

Variables can be linked so that changes to one propagate to another. The `upvar` command links across scopes, and the `trace` command (used in Chapter 17) attaches callbacks that fire when a variable is read, written, or unset. Both mechanisms build on the same variable machinery introduced here.

What You Learned

- `set name value` creates a variable; `set name` reads it.
- `$name` and `${name}` perform variable substitution.
- Arrays hold named elements accessed as `arr(element)`; `array names` lists them.

- `unset` deletes variables.
- Braces create multi-line literal strings with no substitution.

Exercises

1. Build an array `inventory` mapping item names to quantities, then use `foreach` to print each item and quantity.
2. Use `array names` and `lsort` to print the array elements in alphabetical order by key.
3. Use the single-argument `set` to copy one variable into another, then `unset` the original and confirm a subsequent read raises an error (wrap it in `catch`).

Chapter 4: Expressions and Control Flow

Tcl separates "commands" from "expressions." Expressions are the domain of the `expr` command, which evaluates arithmetic, comparison, and logical operators plus a set of math functions. Control flow is provided by the commands `if`, `switch`, `while`, `for`, and `foreach`. This chapter covers all of them, referencing the `expr` man page (expr manual page).

4.1 The expr command

`expr` evaluates an expression and returns the result as a string. Tcl expressions look like C expressions: they support `+ - * / % **`, comparisons

(`< > <= >= == !=`), string comparisons (`eq ne`), logical operators (`&&	

`!), and math functions such as sqrt, int, round, abs, and rand`. The manual page lists the full operator precedence table and the available math functions (expr manual page).

A crucial point: integer division of two integer operands yields an integer (truncated toward zero). To get a floating result, at least one operand must be a floating-point value.

> **Tip:** Always brace the argument to `expr`, as in `expr {$a + $b}`. This
> lets the Tcl compiler produce efficient bytecode and avoids security and
> correctness bugs from double substitution.

4.2 if, switch, and loops

- `if {cond} {body} elseif {cond} {body} else {body}` - the conditions are expressions, so they are passed to `expr` implicitly; brace them.
- `switch -- $value { pattern body ... default body }` - matches `$value` against patterns. The `--` marks the end of options so a value beginning with `-` is not mistaken for an option.
- `while {cond} {body}` - repeats `body` while `cond` is true.
- `for {start} {cond} {next} {body}` - the classic C-style loop.
- `foreach var $list {body}` - iterates over a list; it can also walk several lists in parallel with `foreach a $listA b $listB {body}`.
- `break` exits the innermost loop; `continue` skips to the next iteration.

4.3 The tested script

`ch04_expr.tcl` demonstrates arithmetic, comparison, logical operators, math functions, `if/elseif/else`, `switch`, and all the loop forms:

```
# ch04_expr.tcl — Expressions and control flow

# Arithmetic expressions
set a 10
set b 3
puts "a + b = [expr {$a + $b}]"
puts "a - b = [expr {$a - $b}]"
puts "a * b = [expr {$a * $b}]"
puts "a / b = [expr {$a / $b}] (integer)"
puts "a / b.0 = [expr {$a / ($b * 1.0)}] (float)"
puts "a % b = [expr {$a % $b}] (remainder)"
puts "a ** b = [expr {$a ** $b}] (power)"

# Comparison operators
puts "a > b: [expr {$a > $b}]"
puts "a == b: [expr {$a == $b}]"
puts "a eq '10': [expr {$a eq {10}}]"

# Logical operators
puts "a > 5 && b > 5: [expr {$a > 5 && $b > 5}]"
puts "a > 5 || b > 5: [expr {$a > 5 || $b > 5}]"

# Math functions
puts "sqrt(16) = [expr {sqrt(16)}]"
puts "int(3.7) = [expr {int(3.7)}]"
puts "round(3.5) = [expr {round(3.5)}]"
puts "abs(-5) = [expr {abs(-5)}]"
puts "rand() = [expr {rand()}]"

# if/elseif/else
set score 85
if {$score >= 90} {
    puts "Grade: A"
} elseif {$score >= 80} {
    puts "Grade: B"
} elseif {$score >= 70} {
    puts "Grade: C"
} else {
    puts "Grade: F"
}

# switch
set color "red"
switch -- $color {
    "red"   { puts "Stop" }
    "yellow" { puts "Caution" }
    "green"  { puts "Go" }
    default  { puts "Unknown color" }
}
```

```
# while loop
set i 0
while {$i < 5} {
    puts "while: i=$i"
    incr i
}

# for loop
for {set j 0} {$j < 5} {incr j} {
    puts "for: j=$j"
}

# foreach
set colors {red green blue}
foreach c $colors {
    puts "foreach: $c"
}

# foreach with multiple lists
set names {Alice Bob Carol}
set ages {25 30 35}
foreach n $names a $ages {
    puts "$n is $a years old"
}

# break and continue
for {set k 0} {$k < 10} {incr k} {
    if {$k == 3} { continue }
    if {$k == 7} { break }
    puts "loop: k=$k"
}
```

4.4 Line-by-line walkthrough

- The arithmetic block shows `+`, `-`, `*`, `/`, `%` (remainder), and `**` (power). `expr {$a / $b}` with a=10, b=3 gives 3 (integer division). `expr {$a / ($b * 1.0)}` forces a float by multiplying b by `1.0`, giving `3.3333333333333335`.

- Comparison operators return `0` or `1` (Tcl's boolean strings). `$a eq {10}` is a *string* comparison: it compares the string `"10"` with `"10"` rather than their numeric values.

- Logical `&&` and `		` short-circuit as in C.

- Math functions: `sqrt(16)` returns `4.0`; `int(3.7)` truncates to 3; `round(3.5)` rounds to 4; `abs(-5)` returns 5; `rand()` returns a pseudo-random float in [0, 1).

- `if {$score >= 90} ... elseif ... else` - note every condition and every body is braced. Bracing the condition passes it to `expr` exactly once, efficiently.

- `switch -- $color { ... }` - the `--` ends option processing. Each pattern is followed by its body in braces; `default` catches anything unmatched.

- `while {$i < 5} { ...; incr i }` - `incr` increments a variable (by 1 by default). Forgetting to increment is an infinite loop.

- `for {set j 0} {$j < 5} {incr j} { ... }` - the four parts: init, condition, step, body.
- `foreach c $colors { ... }` - iterates the list `red green blue`.
- `foreach n $names a $ages { ... }` - walks two lists in parallel, pairing Alice/25, Bob/30, Carol/35.
- The `break`/`continue` loop skips k==3 and stops at k==7, so it prints 0,1,2,4,5,6.

> **Warning:** `if` and `while` conditions are expressions. If you forget to
> brace them, `expr` still runs them, but Tcl substitutes variables first -
> which can produce wrong results or, with untrusted input, even let an
> attacker run arbitrary commands. Always brace conditions.

Concepts in Depth

Numeric types and precision

Although every value is logically a string, `expr` distinguishes integer and floating-point arithmetic by the form of the operands. An operation on two integer-looking strings yields an integer (with `/` truncating toward zero and `%` returning the remainder). If any operand looks like a floating-point number (contains a `.` or an exponent, or is produced by a function like `sqrt`), the whole expression is evaluated as floating point. This is why the chapter's example multiplies by `1.0` to force floating division. For high-precision work, note that Tcl uses double-precision (64-bit) floating point; for arbitrary-precision integers, you would reach for the `math::bignum` package from Tcllib rather than the built-in `expr`.

Operator precedence and associativity

`expr` follows C-like precedence: ** binds tightest, then unary minus, then `* / %`, then `+ -`, then comparisons, then eq/ne and string comparisons,

then `&&`, then `		**`. When in doubt, add parentheses - they cost nothing and**

make intent explicit. The eq and ne operators always compare their operands as *strings*; == and `!=` compare them as *numbers* when both look numeric, which can surprise you when one operand is something like `"0"` and the other `"0e0"`. Use eq/ne when you mean string equality, and ==/`!=` only for genuine numeric comparison.

The cost of unbraced expressions

If you write `expr $a + $b` without braces, Tcl substitutes `$a` and `$b`

first, then passes the resulting string to `expr`, which re-parses it. This is slower (no bytecode caching), fragile (a value containing letters could turn a sum into a syntax error), and dangerous (a malicious value could inject commands). Bracing - `expr {$a + $b}` - lets `expr` see the original expression, compile it once, and perform substitution itself in a controlled way. This single habit prevents a large class of bugs.

Control flow as commands

`if`, `while`, `for`, `foreach`, and `switch` are not special syntax - they are ordinary commands that happen to take scripts as arguments. Their conditions are passed to `expr` internally, and their bodies are scripts evaluated in the caller's scope. Because they are commands, you can define your own control-flow constructs with `proc`, and existing ones can be renamed or wrapped (for example, to add tracing to every `puts`).

switch flavors

`switch` supports three matching modes: exact string (the default), glob (with `-glob`), and regular expression (with `-regexp`). The `--` separator is essential whenever the value might begin with a dash, because `switch` otherwise interprets leading-dash words as options. Using `--` unconditionally is a safe habit.

What You Learned

- `expr` evaluates arithmetic, comparison, logical, and math-function expressions; brace its argument.
- Integer division truncates; mix in a float to get a float result.
- eq/ne compare strings; ==/!= compare numerically.
- `if`, `switch`, `while`, `for`, and `foreach` provide control flow.
- `foreach` can iterate multiple lists in parallel.
- `break` and `continue` alter loop execution.

Exercises

1. Write a `for` loop that prints the squares of the numbers 1 through 10.
2. Use `switch` to translate the numbers 1-5 into the words "one" through "five", with a `default` case for out-of-range input.
3. Convert the `while` loop example into a `for` loop that produces identical output.

Chapter 5: Procedures and Scope

Procedures let you package reusable commands. Tcl procedures are created with `proc`, can have default arguments and variable-length argument lists, and interact with variable scope through `global` and `upvar`. This chapter covers all of these, referencing the `proc` man page (proc manual page).

5.1 Defining procedures

The `proc` command creates a new command. Its syntax is `proc name argsList body`. The `argsList` is a list of parameter names; each parameter may have a default value, written as `{name default}`. The special last parameter name `args` collects any extra arguments into a list. The `body` is a script. A procedure returns the result of the last command in its body, unless an explicit `return` is used (proc manual page).

5.2 Variable scope

Tcl has two scopes by default: the global scope (the top level of the script) and the local scope of each procedure. A procedure's body cannot see global variables unless you declare them. Two commands bridge the gap:

- `global varName` - makes a global variable visible inside the current procedure.
- `upvar ?level? otherVar myVar` - links a variable in a caller's scope to a local name, enabling call-by-reference.

5.3 The tested script

`ch05_proc.tcl` shows default arguments, `args`, implicit return, `global`, `upvar` for swapping, recursion, and `return -code error`:

```
# ch05_proc.tcl — Procedures and variable scope

# Basic procedure
proc greet {name} {
    return "Hello, $name!"
}
puts [greet "World"]

# Procedure with default arguments
proc power {base {exp 2}} {
    return [expr {$base ** $exp}]
}
puts "2^3 = [power 2 3]"
puts "5^2 = [power 5]"

# Procedure with variable args
proc sum_all {args} {
    set total 0
    foreach n $args {
        set total [expr {$total + $n}]
    }
    return $total
}
puts "sum: [sum_all 1 2 3 4 5]"

# Return value (implicit - last command result)
proc square {x} {
    expr {$x * $x}
}
puts "square(7) = [square 7]"

# Global variables
set counter 0

proc increment {} {
    global counter
    incr counter
}

increment
increment
increment
puts "Counter: $counter"

# upvar: pass variable by reference
proc swap {aVar bVar} {
    upvar 1 $aVar a
    upvar 1 $bVar b
    set tmp $a
    set a $b
```

```
    set b $tmp
}

set x "first"
set y "second"
puts "Before: x=$x y=$y"
swap x y
puts "After: x=$x y=$y"

# Recursive procedure
proc factorial {n} {
    if {$n <= 1} {
        return 1
    }
    return [expr {$n * [factorial [expr {$n - 1}]]}]
}
puts "5! = [factorial 5]"

# Procedure with return -code
proc check_positive {n} {
    if {$n < 0} {
        return -code error "number must be positive"
    }
    return "OK: $n"
}
if {[catch {check_positive 5} result]} {
    puts "Error: $result"
} else {
    puts $result
}
```

5.4 Line-by-line walkthrough

- `proc greet {name} { return "Hello, $name!" }` - the simplest procedure: one parameter, explicit return.
- `proc power {base {exp 2}} { return [expr {$base ** $exp}] }` - exp has a default value of 2. Calling `power 5` uses the default; `power 2 3` uses the supplied value.
- `proc sum_all {args} { ... }` - the `args` parameter collects all remaining arguments into a list, so `sum_all 1 2 3 4 5` sums five numbers.
- `proc square {x} { expr {$x * $x} }` - no explicit `return`; the result of the last command (`expr`) becomes the procedure's return value.
- `set counter 0` defines a global variable. `proc increment {} { global counter; incr counter } declares it global so incr` can modify it. Each call increments the shared counter.
- `proc swap {aVar bVar} { upvar 1 $aVar a; upvar 1 $bVar b; ... }` - `upvar` links the caller's variables (named by the *values* of `aVar` and `bVar`) to local names a and b. Swapping a and b therefore swaps the caller's variables. This is how you pass variables by reference in Tcl.
- `proc factorial {n} { if {$n <= 1} { return 1 }; return [expr {$n * [factorial [expr {$n - 1}]]}] } - a recursive procedure. The base case n <= 1 returns 1; otherwise it returns n times the factorial of n-1`.

- `proc check_positive {n} { ...; return -code error "number must be positive" } - return -code error` raises an error that propagates to the caller. The caller uses `catch {check_positive 5} result` to intercept it: `catch` returns a non-zero code if an error occurred and stores the message in `result`.

> **Note:** `upvar 1` refers to the immediate caller. `upvar #0` refers to the
> global scope. The `upvar` mechanism is also how procedures like `incr` and
> `lappend` can modify variables passed to them by name.

Concepts in Depth

Argument lists as values

A procedure's argument list is itself a Tcl list, which means defaults and `args` compose predictably. Each element is either a plain name (a required argument) or a two-element list `{name default}` (an optional argument with a default). The special name `args`, if it appears last, collects all remaining positional arguments into a single list. You cannot have a required argument after an optional one (Tcl raises an error at definition time), because that would be ambiguous at call time.

Call frames and the call stack

Each call to a procedure creates a new *call frame* (a variable scope) on top of the global scope. A procedure can see its own local variables and (via `global`) global variables, but not the locals of its caller - that is what `upvar` is for. `upvar 1` links to the immediate caller; `upvar 2` to the caller's caller; and `upvar #0` links to the global scope. This explicit stack addressing is how `incr`, `lappend`, `dict set`, and other commands modify variables passed to them by name without needing real pointers.

Recursion and the stack limit

Tcl, like most scripting languages, has a finite C stack, so unbounded recursion will eventually crash with a stack overflow. The `factorial` example is fine for small `n`, but deep recursion (tens of thousands of frames) is not. For iterative problems that *look* recursive, consider converting to an explicit loop, or use the `coroutine` and `yield` mechanism introduced in 8.6 for cooperative, heap-based state machines.

Return codes and tail calls

Every Tcl command produces a *result code*: `ok` (normal), `error` (an exception), `return` (a request to return from the current procedure), `break`, and `continue`. The `return -code` option lets a procedure produce

one of these non-ok codes, which is how `check_positive` in the example raises an error that `catch` then intercepts. This code mechanism is also the basis for implementing custom control structures that participate in `break` and `continue` propagation.

Default scoping and the global trap

Beginners often expect a procedure to see variables defined at the top level. It cannot, unless those variables are declared `global`. This is intentional: it keeps procedures self-contained and testable. If you find yourself declaring many globals, that is usually a signal to refactor the shared state into a dictionary or namespace variable instead.

What You Learned

- `proc name args body` defines a command; default arguments use `{name default}`.
- `args` as the last parameter collects extra arguments into a list.
- A procedure returns its last command's result unless `return` is used.
- `global` exposes global variables inside a procedure.
- `upvar` links a caller's variable to a local name, enabling call-by-reference.
- `return -code error` raises an error; `catch` intercepts it.

Exercises

1. Write a procedure `max {a b}` that returns the larger of two numbers using `expr` and `if`.
2. Rewrite `swap` using a single `upvar` call to a temporary variable and verify it still works.
3. Write a recursive procedure to compute the nth Fibonacci number, and add a memoization cache stored in a global array.

Chapter 6: Strings, Lists, and Dictionaries

Tcl's three core data structures are all, at heart, strings: the string, the list, and the dictionary. This chapter covers the `string`, `list`, and `dict` commands, which let you manipulate text, ordered sequences, and key/value maps. References: the `string` man page

(string manual page),
the `list` man page
(list manual page), and
the `dict` man page
(dict manual page).

6.1 Strings

A string is the fundamental Tcl value. The `string` command has many subcommands: `length`, `toupper`, `tolower`, `reverse`, `index`, `range`, `first`, `last`, `replace`, `match` (glob matching), `map`, `is` (type testing), `repeat`, and `trim`/`trimleft`/`trimright` (string manual page).

6.2 Lists

A list is an ordered collection of elements, each of which is itself a string. Lists are written with whitespace-separated words (when the words are simple) or built with commands like `list`, `lappend`, `linsert`, and `lreplace`. Useful subcommands include `llength`, `lindex`, `lrange`, `lsort`, `lreverse`, `lsearch`, `lassign`, `lmap` (Tcl 8.6+), `join`, and `split` (list manual page).

> **Tip:** To safely build a list, use the `list` command rather than typing words separated by spaces. `list "a b" c` produces a two-element list (`{a b} c`), whereas typing `"a b" c` as a literal would be ambiguous.

6.3 Dictionaries

A dictionary (introduced in Tcl 8.5) is an ordered map from keys to values, implemented as a flat, even-length list of alternating keys and values. Unlike arrays, a dictionary is a first-class value: you can pass it to procedures, store it in a variable, or nest it. The `dict` command provides `create`, `get`, `set`, `size`, `keys`, `exists`, `for`, `merge`, and more (dict manual page).

6.4 The tested script

`ch06_strings.tcl` exercises the `string`, `list`, and `dict` commands:

```
# ch06_strings.tcl — Strings, lists, and dictionaries

# --- STRING COMMAND ---
set s "Hello, World!"
puts "String: $s"
puts "Length: [string length $s]"
puts "Upper: [string toupper $s]"
puts "Lower: [string tolower $s]"
puts "Reverse: [string reverse $s]"
puts "Index 0: [string index $s 0]"
puts "Range 0-4: [string range $s 0 4]"
puts "First 'o': [string first o $s]"
puts "Last 'o': [string last o $s]"
puts "Replace: [string replace $s 7 11 {Tcl}]"

# string match (glob patterns)
puts "match 'H*': [string match {H*} $s]"
puts "match 'hello': [string match -nocase {hello*} $s]"

# string map
puts "map: [string map {Hello Hi World Earth} $s]"

# string is
set num "12345"
puts "'$num' is integer: [string is integer $num]"
set word "hello"
puts "'$word' is alpha: [string is alpha $word]"

# string repeat
puts "repeat: [string repeat {abc-} 3]"

# string trim
set padded "   spaced   "
puts "trim: '[string trim $padded]'"
puts "trimleft: '[string trimleft $padded]'"
puts "trimright: '[string trimright $padded]'"

# --- LIST COMMAND ---
set fruits {apple banana cherry date elderberry}
puts "List: $fruits"
puts "Length: [llength $fruits]"
puts "Index 0: [lindex $fruits 0]"
puts "Index 2: [lindex $fruits 2]"
puts "Range 1-3: [lrange $fruits 1 3]"

# List operations
lappend fruits fig
puts "After lappend: $fruits"

set sorted [lsort $fruits]
```

```
puts "Sorted: $sorted"

set reversed [lreverse $fruits]
puts "Reversed: $reversed"

# lsearch
set idx [lsearch $fruits "cherry"]
puts "Index of cherry: $idx"

# linsert
set newList [linsert $fruits 1 "apricot"]
puts "After insert at 1: $newList"

# lreplace
set replaced [lreplace $fruits 0 0 "avocado"]
puts "After replace 0: $replaced"

# lassign
lassign {10 20 30} a b c
puts "lassign: a=$a b=$b c=$c"

# lmap (Tcl 8.6+)
set squares [lmap x {1 2 3 4 5} {expr {$x * $x}}]
puts "Squares: $squares"

# join and split
set words {one two three four}
puts "join: [join $words {, }]"
set csv "a,b,c,d"
puts "split: [split $csv {,}]"

# --- DICTIONARY COMMAND ---
set person [dict create name "Alice" age 30 city "NYC"]
puts "Dict: $person"
puts "Name: [dict get $person name]"
puts "Age: [dict get $person age]"
puts "Size: [dict size $person]"
puts "Keys: [dict keys $person]"

# Add/update
dict set person email "alice@example.com"
puts "After set email: [dict get $person email]"

# Check existence
puts "Has 'name': [dict exists $person name]"
puts "Has 'phone': [dict exists $person phone]"

# Iterate
dict for {key value} $person {
    puts "  $key: $value"
```

```
}

# dict merge
set extra [dict create phone "555-1234" age 31]
set merged [dict merge $person $extra]
puts "Merged age: [dict get $merged age]"
puts "Merged phone: [dict get $merged phone]"
```

6.5 Line-by-line walkthrough

String section

- `string length $s` returns the number of characters (not bytes).
- `string toupper/tolower/reverse` produce a new string.
- `string index $s 0` returns the character at position 0 (H).

- `string range $s 0 4` returns characters 0 through 4 (`Hello`).
- `string first o $s` returns the index of the first `o` (4); `string last` returns the last occurrence.
- `string replace $s 7 11 {Tcl}` replaces characters 7-11 with `Tcl`.
- `string match {H*} $s` does glob matching: `H*` matches any string starting with `H`. The `-nocase` option makes it case-insensitive.
- `string map {Hello Hi World Earth} $s` performs simultaneous substitutions: every `Hello` becomes `Hi` and every `World` becomes `Earth`.
- `string is integer $num` tests whether a string is a valid integer; `string is alpha` tests for letters only.
- `string repeat {abc-} 3` repeats the pattern three times.
- `string trim $padded` removes leading and trailing whitespace; `trimleft`/`trimright` remove from one side only. An optional second argument specifies the characters to trim.

List section

- `set fruits {apple banana cherry date elderberry}` - a five-element list.
- `llength` returns 5.
- `lindex $fruits 0` returns `apple`; `lindex $fruits 2` returns `cherry`.
- `lrange $fruits 1 3` returns elements 1 through 3 as a new list.
- `lappend fruits fig` adds `fig` to the end of the list variable.
- `lsort` returns a sorted copy; `lreverse` returns a reversed copy.
- `lsearch $fruits "cherry"` returns the index of the first matching element.
- `linsert $fruits 1 "apricot"` returns a new list with `apricot` inserted at position 1.
- `lreplace $fruits 0 0 "avocado"` returns a new list with element 0 replaced.
- `lassign {10 20 30} a b c` assigns successive elements to variables.
- `lmap x {1 2 3 4 5} {expr {$x * $x}}` applies an expression to each element, returning `{1 4 9 16 25}`.
- `join $words {, }` turns a list into a string with a delimiter; `split` does the reverse, splitting a string into a list.

Dictionary section

- `dict create name "Alice" age 30 city "NYC"` builds a dictionary.
- `dict get $person name` returns the value for a key.
- `dict size` returns the number of key/value pairs.
- `dict keys` returns the list of keys.
- `dict set person email "alice@example.com"` adds or updates a key in the variable `person`.

- `dict exists $person name` tests whether a key is present.
- `dict for {key value} $person { ... }` iterates over the dictionary.
- `dict merge $person $extra` merges two dictionaries; where keys overlap, the second dictionary's value wins.

> **Note:** Prefer `dict` over arrays for new code. Dictionaries are ordered,
> passable values, and nest naturally, while arrays are tied to variable scope
> and cannot be passed by value.

Concepts in Depth

Strings, lists, and dicts: one model, three views

Because every value is a string, a list is just a string with a particular quoting structure, and a dictionary is just a list of even length. The `string`, `list`, and `dict` commands are therefore not three unrelated toolkits but three perspectives on the same underlying value. You can pass a list to `string` commands (it is a string), and you can pass a properly formed string to `list` commands (they parse it as a list). The discipline is to know which view you intend: use `list` commands when the value is logically a sequence, and `dict` commands when it is logically a map.

List representation and quoting

When Tcl displays a list, it adds braces or quotes only where needed to make the result re-parse to the same list. So a list containing the single element `a b` is displayed as `{a b}`, while a list of three simple words is displayed as `apple banana cherry` with no braces. This minimal quoting means a list printed with `puts` can usually be read back with `lindex` - a property that makes lists a natural serialization format for simple data.

The cost of repeated modification

Lists in Tcl are values, so `lappend`, `linsert`, and `lreplace` on a variable actually replace the variable's value with a new list. For a handful of elements this is cheap, but appending to a very large list inside a tight loop can be O(n^2) because each append copies the whole list. When you need to build a large list incrementally, accumulate elements and use `lappend` (which Tcl optimizes internally for the common case), or build with `lmap`.

Dictionary ordering and merging

Dictionaries preserve insertion order: `dict for` iterates keys in the order they were added. `dict merge` combines dictionaries left to right; when two dictionaries share a key, the right-hand value wins, which is a concise way to apply overrides to a default configuration. Because dictionaries are values,

merging never mutates its inputs - it returns a new dictionary.

Choosing the right structure

Use a plain string when you have one value; a list when you have an ordered sequence of similar items; a dictionary when you have labeled fields or a mapping; and an array (scope-local) only when you need a mutable, shared lookup table tied to a variable. Nested dictionaries are the idiomatic way to represent structured records, and they compose naturally with `dict get`'s multi-key form (`dict get $d a b c` walks a path of nested keys).

What You Learned

- `string` subcommands manipulate text: length, case, indexing, matching, mapping, type testing, trimming.
- Lists are ordered collections; `lappend`, `lsort`, `lsearch`, `linsert`, `lreplace`, `lassign`, `lmap`, `join`, and `split` operate on them.
- Dictionaries are ordered, first-class key/value maps; `dict create`, `get`, `set`, `keys`, `exists`, `for`, and `merge` operate on them.
- Use the `list` command to build lists safely.

Exercises

1. Write a script that takes a sentence, splits it into words with `split`, sorts them with `lsort`, and joins them back with `join`.
2. Build a dictionary of three people's names and ages, then use `dict for` to print each person older than 25.
3. Use `string map` to censor every occurrence of a forbidden word in a paragraph, replacing it with asterisks.

Chapter 7: File I/O, Errors, and Packages

Real programs read and write files, handle errors gracefully, and reuse packaged code. This chapter covers the `open`, `gets`, `puts`, `read`, and `close` commands for file I/O; the `file` command for file information and path manipulation; the `catch` and `try` commands for error handling; and the `package` command for loading extensions. References: the `file` man page (file manual page) and the `open` man page

(open manual page).

7.1 File input and output

The open command opens a file and returns a channel identifier (a string like `file5`). Its second argument is the access mode: `r` (read), `w` (write, truncating), a (append), or variants with + for read/write (open manual page).

- `puts $channel "text"` writes a line to a channel (or to stdout if no channel is given).
- `read $channel` reads the entire remaining content; `read -nonewline $channel` strips the trailing newline.
- `gets $channel lineVar` reads one line, stores it in `lineVar`, and returns the number of characters (or `-1` at end of file).
- `close $channel` closes the file.

7.2 The file command

`file` has many subcommands for querying and manipulating paths (file manual page):

`exists`, `size`, `type`, `readable`, `writable`, `tail` (basename), `dirname`, `extension`, `rootname`, `join`, and `split`.

7.3 Error handling

Two mechanisms handle errors:

- `catch {script} resultVar` runs `script`; returns `0` on success or a non-zero error code on failure, and stores the result or error message in `resultVar`.
- `try {script} on error {msg opts} {handler} finally {cleanup}` - the Tcl 8.6 structured exception handler. `on error` (or `trap`) catches errors; `finally` always runs. The `throw` command raises a typed error; `trap` matches the error type.

7.4 Packages

The package command loads and provides versioned extensions. `package require Tk` loads Tk; `package provide name version` declares that a package is available; `package names` lists loaded packages.

7.5 The tested script

`ch07_files.tcl` writes, reads (whole and line-by-line), appends, queries file metadata, manipulates paths, and demonstrates `catch`, `try/finally`, `try/on error/finally`, and `throw/trap`:

```
# ch07_files.tcl — File I/O, errors, and packages

# --- FILE I/O ---
# Write to a file
set fh [open "test_output.txt" w]
puts $fh "Line 1: Hello from Tcl"
puts $fh "Line 2: File writing works"
puts $fh "Line 3: Closing now"
close $fh

# Read from a file
set fh [open "test_output.txt" r]
set content [read $fh]
close $fh
puts "File content:\n$content"

# Read line by line
set fh [open "test_output.txt" r]
while {[gets $fh line] >= 0} {
    puts "Read: $line"
}
close $fh

# Append to a file
set fh [open "test_output.txt" a]
puts $fh "Line 4: Appended line"
close $fh

# File information
puts "Exists: [file exists test_output.txt]"
puts "Size: [file size test_output.txt]"
puts "Type: [file type test_output.txt]"
puts "Readable: [file readable test_output.txt]"
puts "Writable: [file writable test_output.txt]"

# Path operations
puts "Basename: [file tail /home/user/file.txt]"
puts "Dirname: [file dirname /home/user/file.txt]"
puts "Extension: [file extension /home/user/file.txt]"
puts "Rootname: [file rootname /home/user/file.txt]"
puts "Join: [file join home user docs file.txt]"
puts "Split: [file split /home/user/file.txt]"

# --- ERROR HANDLING ---
# catch command
set result [catch {expr {1 / 0}} errMsg]
puts "catch result: $result, message: $errMsg"

# try...finally (Tcl 8.6+)
try {
```

```
    set fh [open "test_output.txt" r]
    set data [read $fh]
} finally {
    close $fh
}
puts "try/finally read: [string range $data 0 20]..."

# try...catch...finally
try {
    error "Something went wrong"
} on error {errMsg errOpts} {
    puts "Caught error: $errMsg"
} finally {
    puts "Cleanup runs regardless"
}

# throw and trap (Tcl 8.6+)
try {
    throw {MYERR} "Custom error type"
} trap {MYERR} {msg opts} {
    puts "Custom error caught: $msg"
}

# --- PACKAGES ---
# List loaded packages
puts "Packages: [package names]"

# Check if a package is available
if {![catch {package require Tcl}]} {
    puts "Tcl package version: [package provide Tcl]"
}

# Cleanup
file delete test_output.txt
```

7.6 Line-by-line walkthrough

- `set fh [open "test_output.txt" w]` opens the file for writing (creating or truncating it). `puts $fh "..."` writes lines to the channel. `close $fh` flushes and closes.
- `set fh [open "test_output.txt" r]; set content [read $fh]` reads the whole file into `content`.
- The `while {[gets $fh line] >= 0} { ... }` loop reads one line at a time until end of file, where `gets` returns `-1`.
- `open ... a` opens for append; the new line is added to the end.
- `file exists`, `file size`, `file type`, `file readable`, and `file writable` query the file. `file type` returns `link`, `file`, or `directory`.
- `file tail` returns the last path component; `file dirname` the directory; `file extension` the extension (including the dot); `file rootname` the name without the extension; `file join` builds a native path from parts; `file split` breaks a path into a list of components.
- `set result [catch {expr {1 / 0}} errMsg]` - dividing by zero raises an error. `catch` returns 1 (the error code) and stores `"divide by zero"` in

errMsg.
- `try { ... } finally { close $fh }` - the `finally` block guarantees the file is closed even if reading fails.
- `try { error "Something went wrong" } on error {errMsg errOpts} { ... } finally { ... }` - `error raises an error; on error` catches it, binding the message to `errMsg` and the option dictionary to `errOpts`; `finally` always runs.
- `try { throw {MYERR} "Custom error type" } trap {MYERR} {msg opts} { ... }`
- `throw` raises a *typed* error (a list of error-code words); `trap` matches that type.
- `package names` lists loaded packages; `package require Tcl` loads (or confirms) the Tcl core; `package provide Tcl` returns the version.

> **Warning:** Always close channels you open. If an error occurs between
> open and `close`, the channel leaks. The `try ... finally` idiom is the
> safest pattern; `close` in the `finally` block runs regardless of errors.

> **Tip:** Use `file join` to build paths rather than string concatenation.
> `file join` uses the correct separator for the platform, so your scripts
> stay portable.

Concepts in Depth

Channels, not file handles

Tcl generalizes files into *channels*. A channel is a bidirectional, byte-oriented stream identified by a string (like `file5` or `stdin`). Files, pipes, sockets, and the standard streams are all channels, so the same `puts`, `gets`, `read`, `close`, and `fconfigure` commands work on all of them. This abstraction is why a script that reads from a file can be repointed at a pipe or a network socket with almost no change.

Blocking, buffering, and configuration

By default, channels are blocking: `gets` waits until a line is available, and `read` waits until end of input. The `fconfigure` command changes this: `-blocking 0` makes reads non-blocking (returning immediately), `-buffering` controls how output is flushed, and `-encoding` and `-translation` handle character sets and line endings. When reading text files on a platform with different line endings, `-translation auto` (the default) usually does the right thing; for binary data, use `fconfigure $chan -translation binary`.

The error stack and errorInfo

When an error propagates, Tcl builds a stack trace in the global variable

`errorInfo` (and a machine-readable code in `errorCode`). `catch` and `try` intercept the error before it unwinds the stack. The option dictionary bound by `try ... on error {msg opts}` carries the same information programmatically (via `dict get $opts -errorinfo` and `dict get $opts -errorcode`), which lets you log rich diagnostics or decide whether to re-throw.

try vs. catch

`catch` is the older, simpler mechanism: it returns a code and stores a result. `try` (added in 8.6) is more expressive: it distinguishes error *types* with `trap`, runs `finally` cleanup, and lets you re-throw with `try ... finally plus return -code. As a rule, use try` for new code, especially when you need guaranteed cleanup or typed error handling, and use `catch` for quick-and-dirty "does this command succeed?" checks.

Packages and versioning

`package require name ?version?` loads a package, raising an error if it is unavailable or the version is too old. `package provide name version` declares that the current script supplies a package, which is how libraries advertise themselves. The `tclPkg` mechanism searches a set of directories (controlled by the `auto_path` variable) for `pkgIndex.tcl` files that register packages. Understanding this lookup is the key to distributing your own reusable modules.

What You Learned

- `open` returns a channel; modes are `r`, `w`, `a` (plus `+` variants).
- `puts` writes, `read` reads all, `gets` reads a line, `close` closes.
- `file` subcommands query and manipulate paths portably.
- `catch` is the simple error trap; `try/on error/finally` and `throw/trap` are the structured 8.6 mechanisms.
- `package require` loads extensions; `package provide` declares them.

Exercises

1. Write a script that copies one text file to another, reading line by line, using `try/finally` to guarantee both channels are closed.
2. Use `catch` to safely parse a number the user types; print an error message if it is not a valid integer.
3. Use `file join` and `file split` to take a path apart, change its extension to `.bak`, and rebuild it.

Chapter 8: Regular Expressions

Tcl's regular expression engine supports Advanced Regular Expressions (AREs), a rich superset of POSIX EREs with backreferences, character classes, and non-greedy quantifiers. The two commands are `regexp` (match and capture) and `regsub` (substitute). Reference: the `regexp` man page (regexp manual page).

8.1 regexp

`regexp ?options? pattern string ?matchVar ?submatchVar ...??` returns 1 if the pattern matches, `0` otherwise. If you supply variable names, they are filled with the full match and each parenthesized submatch. Options include `-all` (all matches), `-inline` (return matches as a list), `-indices` (return index pairs), and `-nocase`.

8.2 regsub

`regsub ?options? pattern string subspec outVar` substitutes matches of `pattern` in `string` with `subspec` (which can reference captures like `\1`, `\2`) and stores the result in `outVar`. `-all` replaces every match. It returns the number of substitutions.

8.3 The tested script

`ch08_regex.tcl` demonstrates captures, `-all -inline`, `-indices`, `regsub`, character classes, anchors, backreferences, and splitting:

```
# ch08_regex.tcl — Regular expressions in Tcl

# Basic regexp
set str "Hello123World456"
if {[regexp {(\d+)} $str match num]} {
    puts "First number: $num (full match: $match)"
}

# Multiple captures
set date "2024-03-15"
if {[regexp {(\d{4})-(\d{2})-(\d{2})} $date full year month day]} {
    puts "Year: $year, Month: $month, Day: $day"
}

# regexp -all (find all matches)
set nums {}
foreach {- num} [regexp -all -inline {\d+} $str] {
    lappend nums $num
}
puts "All numbers: $nums"

# regexp -indices
if {[regexp {World} $str match indices]} {
    puts "Match at indices: $indices"
}

# regsub (substitution)
set result [regsub {World} $str "Earth"]
puts "After regsub: $result"

# regsub -all
set text "cat bat rat mat"
set result [regsub -all {[bcr]at} $text "DOG"]
puts "After regsub -all: $result"

# Named captures (Tcl 8.7+; use indexed captures in 8.6)
set email "user@example.com"
if {[regexp {([^@]+)@(.+)} $email -> u d]} {
    puts "User: $u, Domain: $d"
}

# Character classes
puts "[regexp {^[[:alpha:]]+$} {Hello}] (alpha only)"
puts "[regexp {^[[:digit:]]+$} {12345}] (digit only)"
puts "[regexp {^[[:alnum:]]+$} {Hello123}] (alnum)"

# Anchors
puts "[regexp {^Hello} {Hello World}] (starts with Hello)"
puts "[regexp {World$} {Hello World}] (ends with World)"
```

```
# Backslash sequences in ARE
set phone "Call 555-1234 now"
if {[regexp {(\d{3})-(\d{4})} $phone full area number]} {
    puts "Phone: ($area) $number"
}

# regsub with backreferences
set result [regsub {(\w+)@(\w+)} "user@host" {\2!\1}]
puts "Backreference: $result"

# Splitting with regexp
set parts [regexp -inline -all {\S+} "  one   two  three  "]
puts "Split parts: $parts"
```

8.4 Line-by-line walkthrough

- `regexp {(\d+)} $str match num` - matches the first run of digits in `Hello123World456`. `match` gets 123 (the full match) and `num` gets 123

(the first parenthesized capture). Returns 1.
- `regexp {(\d{4})-(\d{2})-(\d{2})} $date full year month day` - captures the year, month, and day from `2024-03-15` using `{4}` and `{2}` repetition counts.
- `regexp -all -inline {\d+} $str` - returns all digit runs as a flat list. The `foreach {- num} [...]` loop ignores the full match (bound to the throwaway variable `-`) and collects the captured numbers.
- `regexp -indices` - stores start/end index pairs instead of substrings.
- `regsub {World} $str "Earth"` - replaces the first `World` with `Earth`.
- `regsub -all {[bcr]at} $text "DOG"` - replaces every `bat`, `cat`, or `rat` with `DOG`. `[bcr]` is a character class.
- `regexp {([^@]+)@(.+)} $email -> u d` - the `->` is a convention for the "full match" variable you don't care about; u captures the user part and d the domain.
- Character classes: `[[:alpha:]]` matches letters, `[[:digit:]]` digits, `[[:alnum:]]` letters or digits. Anchored with ^ (start) and $ (end), `^[[:alpha:]]+$` matches a string that is letters only.
- `regexp {(\d{3})-(\d{4})} $phone full area number` captures an area code and number.
- `regsub {(\w+)@(\w+)} "user@host" {\2!\1}` uses backreferences: `\1` is the user and `\2` the host, producing `host!user`.
- `regexp -inline -all {\S+} " one two three "` - `\S+` matches runs of non-whitespace, effectively splitting the string into words (the `\S` is a backslash sequence for "non-whitespace").

> **Tip:** In Tcl regexps, `\d`, `\w`, `\s` and their uppercase opposites
> work inside AREs. Because backslashes are also Tcl escape characters, brace
> your patterns (`{\d+}`) so the backslashes reach the regexp engine
> literally.

> **Note:** Tcl 8.6 uses indexed captures (`\1`, `\2`). Named captures are
> available in Tcl 8.7+, so the script uses indexed/named-convention variables
> (`->`, `u`, `d`) for portability.

Concepts in Depth

ARE: Tcl's regex flavor

Tcl implements Advanced Regular Expressions (AREs), a rich flavor that includes the usual `.` `*` `+` `?` `|` `(` `)` `[` `]` plus backreferences (`\1`), non-greedy quantifiers (`*?` `+?`), lookahead, and named (in 8.7+)

captures. It also supports POSIX character classes like `[[:alpha:]]` and collating elements. AREs are a superset of POSIX ERE, so most ERE patterns work unchanged, but features such as backreferences make AREs more powerful than strict EREs.

Backslashes: two layers

A regexp pattern is a Tcl word, so it goes through Tcl's substitution pass first. The pattern `{\d+}` is braced, so the backslash reaches the regexp engine literally and `\d` means "a digit." If you wrote `"$\d+"` with quotes, the `\d` would still pass through (backslash substitution turns `\d` into `d` only if `\d` is a recognized escape; since it is not, Tcl leaves it as `\d`), but the safe and unambiguous choice is always to brace patterns. This avoids subtle differences between the two parsing layers.

Greedy vs. non-greedy

Quantifiers are greedy by default: `\d+` matches as many digits as possible. Append `?` to make them lazy: `\d+?` matches as few as possible. Greedy matching is usually what you want for whole-token extraction; lazy matching is useful when a token is delimited (for example, matching a quoted string with `".*?"` to stop at the first closing quote).

regexp vs. regsub vs. string map

Choose the simplest tool that works. `string map` is fastest for fixed, simultaneous literal substitutions. `regsub` is for pattern-based substitution, including backreferences. `regexp` is for extraction and validation. Reaching for a regexp when `string first` or `string match` would do is a common over-engineering mistake; conversely, chaining many `string` operations when one regexp would do is the other extreme.

Performance with -inline and -all

`regexp -all -inline` returns all matches as a single list, which is often cleaner than a `while` loop with `-indices`. For very large inputs, compiling the pattern once with `regexp` is effectively cached, so repeated calls on the same pattern are efficient. If you need to match the same pattern against many strings in a loop, factor the pattern into a variable so it is obvious it is constant.

What You Learned

- `regexp` matches and captures; supply variables for the full match and each submatch.
- `-all -inline` returns all matches as a list; `-indices` returns positions.

- regsub substitutes; -all replaces every match; \1, \2 reference captures.
- Character classes [[:alpha:]], [[:digit:]], [[:alnum:]] and anchors ^/$ constrain matches.
- Brace patterns so backslash sequences reach the engine literally.

Exercises

1. Write a regexp that validates an email address of the form user@host.tld, capturing each part.
2. Use regsub -all to replace every vowel in a sentence with *.
3. Use regexp -all -inline to extract all numbers from a string and sum them with expr.

Chapter 9: Tk Fundamentals

Tk is Tcl's GUI toolkit. It provides windows, widgets, an event loop, and a binding mechanism. This chapter introduces the windowing shell wish, the root window, the wm command for window management, the label and button widgets, the pack geometry manager, and the event loop. Reference: the Tk commands overview (Tk manual page).

9.1 wish and the event loop

When you run a script with wish, Tk is loaded, a root window named . is created, and after the script finishes executing its commands, wish enters the **event loop**. The event loop waits for events (button clicks, key presses, timer expirations) and dispatches them to the bound handlers. The script sets everything up; the event loop keeps the program alive and responsive until the window is closed (Tk manual page).

9.2 The window manager

The wm (window manager) command interacts with the platform window manager. wm title . "My App" sets the title bar text; wm geometry . "300x200" sets the size. wm protocol . WM_DELETE_WINDOW {script} overrides what happens when the user clicks the window's close button - by default the window is

destroyed but `wish` may keep running, so binding `WM_DELETE_WINDOW` to `exit` is the common way to quit cleanly.

9.3 Widgets and the packer

A widget is created by a command named after its class (`label`, `button`, `text`, etc.). The first argument is the widget's **path name**, a dot-separated hierarchy rooted at `.`. For example, `.title` is a child of the root, and `.btn` is another child. Widget options (`-text`, `-font`, `-fg`, `-command`) configure appearance and behavior.

Creating a widget does not display it. A **geometry manager** positions it. `pack .title -pady 20` packs the widget into its parent, with vertical padding. The packer is covered fully in Chapter 11.

9.4 The tested script

`ch09_first_gui.tcl` builds a minimal window with a title label, a button, and a status label:

```
# ch09_first_gui.tcl — Tk fundamentals: first window
# Run with: wish ch09_first_gui.tcl

package require Tk

# Create the main window
wm title . "My First Tk App"
wm geometry . "300x200"

# Add a label
label .title -text "Hello, Tk!" -font {Helvetica 18 bold} -fg "#01696F"
pack .title -pady 20

# Add a button
button .btn -text "Click Me!" -command {
    .lbl configure -text "Button clicked!"
}
pack .btn -pady 10

# Add a status label
label .lbl -text "Waiting..." -fg "gray"
pack .lbl -pady 10

# Bind window close
wm protocol . WM_DELETE_WINDOW { exit }
```

9.5 Line-by-line walkthrough

- `package require Tk` - explicitly loads the Tk package. (`wish` loads it automatically, but stating it makes the script's dependency clear and lets it run under `tclsh` with a loaded Tk.)
- `wm title . "My First Tk App"` - sets the root window's title.
- `wm geometry . "300x200"` - sets it to 300 by 200 pixels.

- `label .title -text "Hello, Tk!" -font {Helvetica 18 bold} -fg "#01696F"` - creates a label named `.title` (a child of the root). `-font` takes a list: family, size, and style words. `-fg` sets the foreground (text) color; here a hex color string is used.
- `pack .title -pady 20` - packs the label into the root, top to bottom, with 20 pixels of vertical padding.
- `button .btn -text "Click Me!" -command { .lbl configure -text "Button clicked!" } - creates a button. The -command` option is a script that runs when the button is clicked; here it changes the text of the `.lbl` label.
- `pack .btn -pady 10` - packs the button below the title.
- `label .lbl -text "Waiting..." -fg "gray"` and `pack .lbl -pady 10` - a status label packed below the button.
- `wm protocol . WM_DELETE_WINDOW { exit }` - when the user closes the window, the script runs `exit` instead of just destroying the window, so `wish` terminates cleanly.

> **Note:** Widget path names form a tree. The root is `.`. Children of `.` are
> `.a`, `.b`. Children of `.a` are `.a.x`, `.a.y`. The path determines the
> widget's parent for packing/gridding.

Concepts in Depth

The event loop and blocking

A Tk program is event-driven: it sets up widgets and bindings, then hands control to the event loop, which sleeps until an event arrives, dispatches it to the matching binding, and repeats. This means you must never run a long-running command in a callback, because it blocks the event loop and freezes the UI. Long work belongs in a coroutine, in small chunks scheduled with `after`, or (for genuinely blocking I/O) in a separate thread via the `thread` package. The `after` command, introduced for animation in Chapter 13, is also the primary tool for cooperative multitasking.

The widget path tree

Widget names are paths in a tree rooted at `.`. Creating `button .f.b` makes a button whose parent is the frame `.f`; the frame must exist first. The path determines how geometry managers nest children: you pack or grid `.f.b` *inside* `.f`, not inside `.`. This tree structure mirrors the visual containment hierarchy and keeps naming predictable as applications grow.

Configuration options and the option database

Every widget accepts a large common set of options (`-bg`, `-fg`, `-font`,

`-relief`, `-bd`, `-width`, `-height`, etc.). You can set them at creation (`button .b -text Hi`), change them later (`configure`), and read them (`cget`). Tk also maintains an *option database* (set with `option add`) that provides defaults, so you can theme an entire application by setting a few class defaults rather than configuring each widget individually. The themed (`ttk::`) widgets, mentioned in the conclusion, take this theming model further.

wish vs. tclsh and package require Tk

Under `wish`, Tk is loaded automatically and the event loop starts after the script runs. Under `tclsh`, neither is true. Including `package require Tk` makes a script portable to both interpreters (though `tclsh` will still not start the event loop without `vwait forever`). Stating the dependency explicitly also documents the script's needs.

Destroying widgets and memory

`destroy .f` destroys a widget and all its children, releasing their resources. Variables bound to a destroyed widget via `-textvariable` are *not* unset, but the widget no longer updates them. Keeping widget lifetimes and variable lifetimes in sync is a recurring concern in larger applications, which is one reason the project chapters use global state carefully.

What You Learned

- `wish` loads Tk, creates the root window `.`, and runs the event loop.
- `wm title` and `wm geometry` configure the root window.
- Widgets are created by class commands (`label`, `button`) with a path name and options.
- `pack` places widgets; the script runs once, then the event loop takes over.
- `wm protocol . WM_DELETE_WINDOW { exit }` quits cleanly on close.

Exercises

1. Change the title label's font and color, and observe the effect.
2. Add a second button that resets the status label to "Waiting...".
3. Use `wm geometry . "400x300"` and add `-side left` to one of the `pack` calls to see how the layout changes.

Chapter 10: Widgets

Tk provides a rich set of standard widgets. This chapter tours the most common ones - `label`, `button`, `checkbutton`, `radiobutton`, `entry`, `text`, `listbox`, `spinbox`, and `scale` - showing how each is created, configured, and linked to variables. Reference: the Tk commands overview (Tk manual page).

10.1 Widget basics

Every widget command shares a common set of options (`-text`, `-font`, `-fg`, `-bg`, `-relief`, `-bd`, `-width`, `-height`) documented under the `options` man page. Two patterns recur:

- `-textvariable` links a widget's displayed text to a Tcl variable. Changing the variable updates the widget, and (for entry widgets) editing the widget updates the variable.
- `-command` attaches a script that runs on a user action (a button click, a scale drag).

10.2 The tested script

`ch10_widgets.tcl` is a showcase of nine widget types:

```
# ch10_widgets.tcl — Tk widgets: labels, buttons, entries, text, listbox
# Run with: wish ch10_widgets.tcl

package require Tk
wm title . "Widget Showcase"
wm geometry . "400x500"

# --- LABEL ---
label .lbl -text "Label: static text" -font {Helvetica 10}
pack .lbl -pady 5

# Dynamic label
set count 0
label .counter -textvariable count -font {Helvetica 14 bold} -fg "blue"
pack .counter -pady 5
button .inc -text "Increment" -command { incr count }
pack .inc -pady 2

# --- BUTTON ---
button .btn -text "Click Me" -command {
    .lbl configure -text "Button was clicked!"
}
pack .btn -pady 5

# --- CHECKBUTTON ---
set check_val 0
checkbutton .cb -text "Enable feature" -variable check_val
pack .cb -pady 5

# --- RADIOBUTTON ---
set radio_val "option1"
frame .rf
radiobutton .rf.r1 -text "Option 1" -variable radio_val -value "option1"
radiobutton .rf.r2 -text "Option 2" -variable radio_val -value "option2"
radiobutton .rf.r3 -text "Option 3" -variable radio_val -value "option3"
pack .rf.r1 .rf.r2 .rf.r3 -side left -padx 5
pack .rf -pady 5

# --- ENTRY ---
frame .ef
label .ef.l -text "Enter name:"
entry .ef.e -textvariable entry_name -width 20
pack .ef.l .ef.e -side left -padx 5
pack .ef -pady 5
button .ef.b -text "Show" -command {
    tk_messageBox -message "You entered: $entry_name"
}
pack .ef.b -pady 2

# --- TEXT WIDGET ---
```

```
text .txt -width 40 -height 5 -wrap word
.txt insert end "Text widget: multi-line\ntext editing area.\n"
pack .txt -pady 5

# --- LISTBOX ---
listbox .lb -height 4 -selectmode single
.lb insert 0 "Item 1" "Item 2" "Item 3" "Item 4"
pack .lb -pady 5
button .lb_btn -text "Show Selection" -command {
    set sel [.lb curselection]
    if {$sel ne ""} {
        tk_messageBox -message "Selected: [.lb get $sel]"
    }
}
pack .lb_btn -pady 2

# --- SPINBOX ---
spinbox .sb -from 1 -to 100 -textvariable spin_val -width 5
set spin_val 50
pack .sb -pady 5

# --- SCALE ---
scale .sc -from 0 -to 100 -orient horizontal -label "Volume" -variable scale_val
set scale_val 50
pack .sc -pady 5

wm protocol . WM_DELETE_WINDOW { exit }
```

10.3 Line-by-line walkthrough

- `label .lbl -text "Label: static text"` - a static label.
- `label .counter -textvariable count` - a label whose text is *bound* to the variable `count`. The button `.inc` runs `incr count`, which updates the variable, which updates the label automatically.
- `button .btn -command { .lbl configure -text "Button was clicked!" }` - the `-command` script uses `configure` to change a widget option at runtime.
- `checkbutton .cb -text "Enable feature" -variable check_val` - a checkbox bound to `check_val`, which is `0` (off) or `1` (on).
- `radiobutton .rf.r1 -variable radio_val -value "option1"` - radio buttons share a single variable; selecting one sets the variable to that button's `-value`.
- `entry .ef.e -textvariable entry_name -width 20` - a single-line text field bound to `entry_name`. The "Show" button reads `$entry_name` and displays it in a message box.
- `text .txt -width 40 -height 5 -wrap word` - a multi-line text widget. `.txt insert end "..."` appends text at the end. The text` widget supports rich editing, tags, and marks (see Chapters 14 and 16).
- `listbox .lb -height 4 -selectmode single` then `.lb insert 0 "Item 1" ... - a list of selectable strings. .lb curselection` returns the selected index; `.lb get $sel` returns the text.

- `spinbox .sb -from 1 -to 100 -textvariable spin_val` - a numeric spinner.
- `scale .sc -from 0 -to 100 -orient horizontal -label "Volume"` - a slider.

> **Tip:** Binding widgets to variables with `-textvariable` and
> `-variable` is the idiomatic Tk way to keep the UI and your data in sync
> without manual `configure` calls.

Concepts in Depth

The -variable contract

The `-textvariable` and `-variable` options establish a two-way link between a widget and a Tcl variable: assigning the variable updates the widget, and (for editable widgets) editing the widget updates the variable. This is Tk's data-binding mechanism, and it is the backbone of the project chapters. The link is by *name*, so the variable must exist in the scope where the widget reads it; for widgets created at the top level, that is the global scope, which is why the scripts use plain `set` (global) variables.

Widget classes and the option database

Each widget belongs to a class (e.g., `Button`, `Entry`). The option database can set defaults per class (`option add *Button.background grey`), so a single line can restyle every button. Understanding this layer explains why some options appear to inherit values you never set: they come from the option database, the platform theme, or the widget's class defaults.

The text widget's power

The `text` widget is far more than a multi-line entry: it supports tags (regions with styling and bindings), marks (named positions), embedded windows and images, peer widgets (multiple views of one buffer), and an undo/redo stack (`-undo 1`). Chapters 14 and 16 exploit tags for selection and `<<Modified>>` for change tracking, but the same machinery can implement syntax highlighting, hyperlinks, and rich-text editing. Mastering the text widget is one of the highest-leverage skills in Tk.

listbox vs. ttk::treeview

The `listbox` is simple and good for a flat list of strings. For multi-column, hierarchical, or styled lists, the `ttk::treeview` widget is the modern choice - it supports columns, icons, and a tree structure, and it respects the platform theme. The `listbox` remains useful in this book because it is the most direct illustration of selection and indexing.

Spinbox and scale as bounded inputs

Both `spinbox` and `scale` constrain input to a range. `spinbox` is best for

discrete or small ranges where typing is also acceptable; `scale` is best for continuous values where dragging is natural. Both can be bound to a variable, so reading the current value is just `$var`, and both accept a `-command` that fires on every change, which is useful for live updates.

What You Learned

- `label` and `button` display text and trigger commands.
- `checkbutton` and `radiobutton` bind to variables (boolean / choice).
- `entry` is single-line input; `text` is multi-line.
- `listbox` shows a selectable list; `curselection` and `get` read it.
- `spinbox` and `scale` select numeric values.
- `-textvariable`/`-variable` link widgets to Tcl variables automatically.

Exercises

1. Add a second counter variable and a label bound to it, incremented by a separate button.
2. Replace the `listbox` selection display with code that prints the selected item to the console using `puts`.
3. Make the `scale`'s current value appear in a `label` in real time using `-textvariable`.

Chapter 11: Geometry Managers

A geometry manager positions widgets inside their parent. Tk provides three: `pack`, `grid`, and `place`. Each has its own model. References: the `pack` man page (pack manual page)
and the `grid` man page
(grid manual page).

11.1 pack

`pack` arranges widgets in order along a side of the parent (top, bottom, left, or right). It is simple and good for sequential layouts. Key options: `-side` (top/bottom/left/right), `-fill` (none/x/y/both), `-expand` (0/1, whether to grow into extra space), `-padx`/`-pady` (padding)
(pack manual page).

11.2 grid

`grid` arranges widgets in a rows-and-columns table. It is the best choice for form-like layouts. `grid widget -row r -column c` places a widget; `grid columnconfigure . c -weight 1` makes a column absorb extra horizontal space; `grid rowconfigure . r -weight 1` does the same for rows. `-sticky` controls how a widget attaches to its cell edges (`n s e w` and combinations) (grid manual page).

11.3 place

`place` positions widgets at absolute or relative coordinates. It is powerful but rarely used for whole layouts because it does not adapt to resizing. `-x`/`-y` are absolute pixel coordinates; `-relx`/`-rely` are fractions of the parent size; `-anchor` sets the anchor point.

11.4 The tested script

`ch11_layout.tcl` demonstrates all three managers in three labelframes:

```
# ch11_layout.tcl — Geometry managers: pack, grid, place
# Run with: wish ch11_layout.tcl

package require Tk
wm title . "Layout Managers"
wm geometry . "450x350"

# --- PACK MANAGER ---
# pack arranges widgets in order, top/bottom/left/right
labelframe .pack_frame -relief groove -bd 2 -text "pack"
label .pack_frame.l1 -text "Top" -bg lightblue
label .pack_frame.l2 -text "Bottom" -bg lightyellow
button .pack_frame.b1 -text "Left"
button .pack_frame.b2 -text "Right"

pack .pack_frame.l1 -side top -fill x
pack .pack_frame.l2 -side bottom -fill x
pack .pack_frame.b1 -side left -fill both -expand 1
pack .pack_frame.b2 -side right -fill both -expand 1
pack .pack_frame -fill both -expand 1 -padx 5 -pady 5

# --- GRID MANAGER ---
labelframe .grid_frame -relief groove -bd 2 -text "grid"
label .grid_frame.title -text "Grid Layout" -font {Helvetica 10 bold}
label .grid_frame.l1 -text "Name:"
entry .grid_frame.e1 -width 15
label .grid_frame.l2 -text "Email:"
entry .grid_frame.e2 -width 15
label .grid_frame.l3 -text "Phone:"
entry .grid_frame.e3 -width 15
button .grid_frame.save -text "Save"
button .grid_frame.cancel -text "Cancel"

grid .grid_frame.title - -sticky w -padx 5
grid .grid_frame.l1 .grid_frame.e1 -sticky w -padx 5 -pady 2
grid .grid_frame.l2 .grid_frame.e2 -sticky w -padx 5 -pady 2
grid .grid_frame.l3 .grid_frame.e3 -sticky w -padx 5 -pady 2
grid .grid_frame.save .grid_frame.cancel -padx 5 -pady 5

pack .grid_frame -fill x -padx 5 -pady 5

# --- PLACE MANAGER ---
labelframe .place_frame -relief groove -bd 2 -height 80 -text "place"
label .place_frame.abs -text "Absolute" -bg lightgreen
label .place_frame.rel -text "Relative" -bg lightcoral

place .place_frame.abs -x 10 -y 20
place .place_frame.rel -relx 0.7 -rely 0.5 -anchor center

pack .place_frame -fill x -padx 5 -pady 5
```

```
# Grid columnconfigure for resizing
grid columnconfigure .grid_frame 1 -weight 1

wm protocol . WM_DELETE_WINDOW { exit }
```

11.5 Line-by-line walkthrough

pack section

- `labelframe .pack_frame -relief groove -bd 2 -text "pack"` - a container with a border and a title.
- `pack .pack_frame.l1 -side top -fill x` - pack the "Top" label along the top, stretching it horizontally to fill the width.
- `pack .pack_frame.l2 -side bottom -fill x` - the "Bottom" label along the

bottom.

- `pack .pack_frame.b1 -side left -fill both -expand 1` - the "Left" button along the left side, filling both directions and expanding into spare space.
- `pack .pack_frame.b2 -side right -fill both -expand 1` - the "Right" button mirrors it on the right.

grid section

- `grid .grid_frame.title - -sticky w` - the title spans two columns (a `-` placeholder means "span the previous widget's column"); `-sticky w` glues it to the left (west) edge.
- `grid .grid_frame.l1 .grid_frame.e1 -sticky w -padx 5 -pady 2` - places a label and entry side by side in one row.
- `grid .grid_frame.save .grid_frame.cancel` - the Save and Cancel buttons in the last row.
- `grid columnconfigure .grid_frame 1 -weight 1` - column 1 (the entry column) gets all extra horizontal space, so entries grow when the window resizes.

place section

- `place .place_frame.abs -x 10 -y 20` - absolute placement at pixel (10, 20).
- `place .place_frame.rel -relx 0.7 -rely 0.5 -anchor center` - relative placement at 70% across, 50% down, anchored at its center, so it stays proportional when the parent resizes.

> **Warning:** Never mix `pack`, `grid`, and `place` on widgets that share the
> same parent. Using two managers in the same parent causes Tk to block
> indefinitely while each manager waits for the other. Use one manager per
> parent, or use separate `frame` containers.

Concepts in Depth

How pack works

pack maintains a *parcel* for each child it manages. Each packed widget claims a side of the remaining space, in the order the `pack` commands run. `-fill` controls how the widget fills its parcel (none, x, y, or both); `-expand 1` tells the packer to give this widget a share of any extra space when the parent grows. The interaction of order, side, fill, and expand is what produces a layout, and small changes can shift the whole arrangement - which is why `grid` is often clearer for forms.

How grid works

`grid` divides the parent into rows and columns. Each widget occupies one

cell (or a span of cells via `-` placeholders or `-columnspan`/`-rowspan`). `-sticky` controls which cell edges the widget attaches to: `-sticky nsew` makes it fill the cell completely. `grid columnconfigure . c -weight 1` and `grid rowconfigure . r -weight 1` distribute extra space proportionally to the weights; a column with weight 0 keeps its natural size. Weights are the key to resize-friendly layouts, and the calculator project in Chapter 15 relies on them to make buttons grow evenly.

place: precision without flexibility

`place` positions a widget at absolute or relative coordinates, optionally with a fixed size and an anchor. It does not adapt to parent resizing or content changes, so it is best for overlays, dragged windows, or tightly controlled layouts (like a custom dialog). Using `place` for a whole application is almost always a mistake because every resize must be handled manually.

Containers: frame and labelframe

`frame` and `labelframe` are invisible (or lightly decorated) containers that group children. They let you apply different geometry managers to different regions: pack the frames, and grid (or pack) inside each frame independently. This compartmentalization is the standard way to build complex layouts without triggering the "one manager per parent" deadlock.

The one-manager-per-parent rule

Tk allows `pack`, `grid`, and `place` to coexist in the same application, but not in the same parent. Two managers competing for the same child's geometry cause the parent to enter an infinite negotiation loop and the application to hang. The fix is always structural: introduce a `frame` and use a different manager inside it.

What You Learned

- `pack` arranges widgets sequentially by side; `-fill` and `-expand` control growth.
- `grid` arranges widgets in a table; `columnconfigure`/`rowconfigure` with `-weight` distribute extra space; `-sticky` attaches to cell edges.
- `place` uses absolute (`-x`/`-y`) or relative (`-relx`/`-rely`) coordinates.
- Use one geometry manager per parent to avoid deadlocks.

Exercises

1. Rebuild the grid form with three label/entry rows and a row of two buttons

spanning two columns each.

2. Convert the pack section to use `-side left` for all four widgets and observe the layout.

3. Add `grid rowconfigure` calls so the form rows grow vertically when the window is enlarged.

Chapter 12: Events and Bindings

User interaction in Tk is delivered as **events**: mouse clicks, motion, key presses, and more. The `bind` command attaches a script to an event for a widget (or globally). Reference: the `bind` man page (bind manual page).

12.1 Event patterns

An event pattern is enclosed in angle brackets, e.g. `<Button-1>` (left mouse button press), `<Motion>` (mouse movement), `<Key-Return>` (the Enter key), `<Double-Button-1>` (double click), `<Control-Key-c>` (Ctrl+C). Within a bound script, **percent substitutions** insert event details: `%x` and `%y` are the pointer coordinates, `%A` is the character typed, `%W` is the widget path (bind manual page).

12.2 bind and bind all

`bind widget <pattern> script` binds `script` to the event for `widget`. `bind all <pattern> script` binds to *every* widget, which is useful for global shortcuts like help keys. `bindtags` controls which binding tags a widget responds to (every widget has the `all` tag by default).

12.3 The tested script

`ch12_events.tcl` binds clicks, motion, keys, double-clicks, and a global help key:

```
# ch12_events.tcl — Events and bindings
# Run with: wish ch12_events.tcl

package require Tk
wm title . "Events and Bindings"
wm geometry . "400x300"

# Label to display event info
label .info -text "Try: click, key press, mouse motion" -fg "gray"
pack .info -pady 10

label .status -text "" -font {Helvetica 12} -fg "blue"
pack .status -pady 10

# Canvas to capture events
canvas .c -width 300 -height 150 -bg "white" -relief sunken -bd 2
pack .c -pady 10

# --- BINDINGS ---
# Button-1 = left mouse click
bind .c <Button-1> {
    set x %x
    set y %y
    .status configure -text "Clicked at ($x, $y)"
    .c create oval [expr {$x-3}] [expr {$y-3}] [expr {$x+3}] [expr {$y+3}] -fill red
}

# Mouse motion
bind .c <Motion> {
    .info configure -text "Mouse at (%x, %y)"
}

# Key bindings (on the main window)
bind . <Key-Return> {
    .status configure -text "Enter key pressed"
}

bind . <Key-Escape> {
    .status configure -text "Escape pressed - press q to quit"
}

bind . <Key-q> {
    exit
}

bind . <Control-Key-c> {
    .status configure -text "Ctrl+C pressed"
}

# Double-click
```

```
bind .c <Double-Button-1> {
    .status configure -text "Double-clicked!"
    .c delete all
}

# Bind to all widgets (bindtags)
bind all <Key-F1> {
    .status configure -text "F1 pressed - Help"
}

# Focus for key events
focus .c

wm protocol . WM_DELETE_WINDOW { exit }
```

12.4 Line-by-line walkthrough

- `bind .c <Button-1> { set x %x; set y %y; .status configure -text "Clicked

at ($x, $y)"; .c create oval [expr {$x-3}] [expr {$y-3}] [expr {$x+3}]
[expr {$y+3}] -fill red } - on a left click in the canvas, %x/%y` give
the pointer position relative to the canvas, the status label is updated,
and a small red dot is drawn.
- bind .c <Motion> { .info configure -text "Mouse at (%x, %y)" } - tracks
the pointer as it moves.
- bind . <Key-Return> { ... } - the Enter key, bound on the root window ..
- bind . <Key-Escape> { ... } - the Escape key.
- bind . <Key-q> { exit } - pressing q quits. Note this is bound on .,
so it works only when a descendant of . has the keyboard focus.
- bind . <Control-Key-c> { ... } - Ctrl+C.
- `bind .c <Double-Button-1> { .status configure -text "Double-clicked!"; .c
delete all }` - a double-click clears the canvas.
- bind all <Key-F1> { ... } - F1 shows help on *any* widget, because all
applies to every widget.
- focus .c - directs keyboard events to the canvas so key bindings work
without first clicking it.
> **Tip:** Percent substitutions like %x, %y, %A, and %W are the bridge
> between the raw event and your script. They are substituted *before* the
> script runs, so you can use them directly in commands.
> **Warning:** Key bindings on . only fire when a descendant has the
> keyboard focus. Call focus $widget (or make the widget -takefocus 1 and
> click it) to ensure your key bindings are active.

Concepts in Depth

Event patterns in detail

An event pattern can be a single event (<Button-1>), a sequence
(<Double-Button-1>), or a combination with modifiers (<Control-Shift-Key-A>).
Key events use keysyms (Key-Return, Key-Escape, Key-a); the keysyms
man page lists them all. Mouse events include ButtonPress, ButtonRelease,
Motion, Enter, Leave, and the mouse wheel (<MouseWheel>). Sequences in
angle brackets match the whole sequence; a bare keysym without brackets (e.g.,
bind . q {exit}) also works for simple key presses.

Percent substitutions

Within a binding script, percent sequences are replaced before the script
runs: %x/%y (pointer x/y relative to the widget), %X/%Y (relative to
the screen), %A (the character typed), %K (the keysym), %W (the widget

path that received the event), %b (mouse button number), and %s (modifier state). These let one binding script serve many events and widgets. If you need a literal %, write %%.

Binding tags and bind all

Every widget has an ordered list of *binding tags*. By default it is {widgetPath widgetClass all topLevel}. When an event arrives, Tk checks each tag in order and runs the first matching binding. bind all <pattern> adds a binding to the all tag, so it applies to every widget - useful for global shortcuts. bindtags lets you reorder or replace a widget's tags, which is how you make a text widget behave like a class plus custom behavior, or how you temporarily disable all bindings.

Focus and keyboard events

Keyboard events are delivered to the widget that has the *keyboard focus`. Without focus, key bindings do not fire. focus $widget sets focus; the -takefocus option controls whether a widget accepts focus on click or traversal (Tab). Forgetting to set focus is the most common reason a key binding "doesn't work."

break and continue in bindings

A binding script can return break to stop further binding tags from being checked for the same event - useful to override default behavior (for example, to make Tab do something custom instead of moving focus). continue skips the rest of the current tag's bindings. These return codes are the same ones used by control-flow commands, reflecting that bindings are scripts.

What You Learned

- Events are written as <Event-Type> patterns; modifiers combine as in <Control-Key-c>.
- bind attaches a script to an event for a widget; bind all is global.
- Percent substitutions (%x, %y, %A, %W) inject event data into the script.
- focus directs keyboard input to a widget.

Exercises

1. Add a binding so that pressing c cycles the canvas background color among three choices.
2. Bind <Button-3> (right click) to clear just the last drawn oval using .c find closest.

3. Bind <Control-Key-a> on the canvas to select (highlight) all drawn ovals.

Chapter 13: Canvas Graphics

The canvas widget is Tk's 2D drawing surface. It can display lines, rectangles, ovals, polygons, arcs, text, and bitmaps, and it supports item tags, item operations, and animation. Reference: the canvas man page (canvas manual page).

13.1 Creating items

Canvas items are created by methods like `create line`, `create rectangle`, `create oval`, `create polygon`, `create arc`, `create text`, and `create bitmap`. Each returns an integer **item id**. Items can carry **tags** (with `-tags`), and many commands accept an id or a tag in place of an id (canvas manual page).

13.2 Item operations

- `coords itemId x1 y1 x2 y2` - get or set an item's coordinates.
- `move itemId dx dy` - move items by a delta.
- `itemconfigure itemId -option value` - change an item's options.
- `delete itemId` - remove items.

13.3 Animation with after

The `after ms script` command schedules `script` to run `ms` milliseconds later, enabling animation. A procedure reschedules itself with `after` to create a loop; `after cancel id` stops it.

13.4 The tested script

`ch13_canvas.tcl` draws a gallery of shapes, uses tags, and animates a bouncing ball:

```
# ch13_canvas.tcl — Canvas graphics
# Run with: wish ch13_canvas.tcl

package require Tk
wm title . "Canvas Graphics"
wm geometry . "500x400"

canvas .c -width 480 -height 350 -bg "white" -relief sunken -bd 2
pack .c -padx 10 -pady 10

# --- SHAPES ---
# Line
.c create line 10 10 100 50 -fill blue -width 2
.c create line 100 50 200 10 -fill blue -width 2

# Arrow line
.c create line 250 10 350 50 -arrow last -fill green -width 2

# Rectangle
.c create rectangle 10 70 100 120 -fill lightblue -outline blue -width 2

# Oval (circle)
.c create oval 120 70 170 120 -fill red -outline darkred -width 2

# Oval (ellipse)
.c create oval 180 70 260 120 -fill yellow -outline orange -width 2

# Polygon (triangle)
.c create polygon 280 120 320 70 360 120 -fill green -outline darkgreen -width 2

# Arc
.c create arc 10 140 100 230 -start 0 -extent 270 -style pieslice -fill cyan -outline blue

# Text
.c create text 200 175 -text "Canvas Text" -font {Helvetica 14 bold} -fill purple

# Bitmap
.c create bitmap 300 175 -bitmap info -foreground blue

# --- TAGS AND ITEM OPERATIONS ---
# Create items with tags
.c create rectangle 10 250 60 300 -fill pink -tags {shape movable}
.c create oval 70 250 120 300 -fill lightgreen -tags {shape movable}
.c create rectangle 130 250 180 300 -fill lightyellow -tags {shape movable}

# Move all tagged items
.c move movable 200 0

# Get item coordinates
set coords [.c coords 1]
```

```
puts "Item 1 coords: $coords"

# Item configuration
.c itemconfigure 1 -fill orange

# --- ANIMATION ---
set anim_id ""
set dx 3
set dy 2

proc animate {} {
    global dx dy anim_id
    set coords [.c coords "moving_ball"]
    if {[llength $coords] == 0} return

    set x1 [lindex $coords 0]
    set y1 [lindex $coords 1]
    set x2 [lindex $coords 2]
    set y2 [lindex $coords 3]

    if {$x1 <= 0 || $x2 >= 480} { set dx [expr {-$dx}] }
    if {$y1 <= 0 || $y2 >= 350} { set dy [expr {-$dy}] }

    .c move "moving_ball" $dx $dy
    set anim_id [after 20 animate]
}

# Create animated ball
.c create oval 200 200 220 220 -fill red -tags {moving_ball}

# Control buttons
frame .controls
button .controls.start -text "Start" -command { animate }
button .controls.stop -text "Stop" -command { after cancel $anim_id }
button .controls.clear -text "Clear" -command { .c delete all }
pack .controls.start .controls.stop .controls.clear -side left -padx 5
pack .controls -pady 5

wm protocol . WM_DELETE_WINDOW { exit }
```

13.5 Line-by-line walkthrough

Shapes

- `create line 10 10 100 50 -fill blue -width 2` - a line segment from (10,10) to (100,50).
- `create line 250 10 350 50 -arrow last` - a line with an arrowhead at the end (`-arrow last`); other values are `none`, `first`, `both`.
- `create rectangle 10 70 100 120 -fill lightblue -outline blue` - a rectangle bounded by opposite corners.
- `create oval 120 70 170 120 -fill red` - a circle (the bounding box is square); `create oval 180 70 260 120` is an ellipse (a wider box).
- `create polygon 280 120 320 70 360 120 -fill green` - a triangle from three points.
- `create arc 10 140 100 230 -start 0 -extent 270 -style pieslice` - a pie-slice arc starting at 0 degrees and sweeping 270 degrees.
- `create text 200 175 -text "Canvas Text" -font {Helvetica 14 bold}` - text

drawn on the canvas.

- `create bitmap 300 175 -bitmap info` - a built-in bitmap.

Tags and operations

- The three shapes near the bottom are created with `-tags {shape movable}`.
- `.c move movable 200 0` - moves *every* item with the `movable` tag 200 pixels right, demonstrating tag-based bulk operations.
- `.c coords 1` - returns the coordinate list of item id 1.
- `.c itemconfigure 1 -fill orange` - changes item 1's fill color.

Animation

- `set dx 3; set dy 2` - velocity deltas.
- `proc animate {}` - reads the ball's coordinates via `` `.c coords "moving_ball"` ``; if the ball hits a wall, the corresponding delta is negated (`set dx [expr {-$dx}]`); then `.c move "moving_ball" $dx $dy` moves it; and `set anim_id [after 20 animate]` schedules the next frame in 20 ms, storing the timer id so it can be cancelled.
- The "Stop" button runs `after cancel $anim_id` to halt the loop.

> **Tip:** Using a tag like `moving_ball` (instead of the numeric id) for the
> animated item makes the code robust: even if you add other items, the
> animation logic still targets the right one by tag.

Concepts in Depth

Item types and coordinates

Each canvas item type has a coordinate convention. Lines, rectangles, and ovals take two corner points (x1 y1 x2 y2). Polygons take any number of points. Arcs take a bounding box plus `-start` and `-extent` in degrees (0 is the 3-o'clock position, angles increase counterclockwise). Text and bitmaps take a single anchor point plus `-anchor`. Understanding these conventions lets you construct items programmatically, as the animation example does by reading and modifying an item's coordinates.

Tags as groups

A tag is a string label that any number of items can share, and an item can have many tags. Most item commands (`move`, `delete`, `itemconfigure`, `raise`, `lower`, `coords`, `scale`) accept a tag wherever they accept an id, and they act on *all* items with that tag. Two special tags always exist: `all` (matches every item) and `current` (the topmost item under the pointer during an event). Tags are the canvas's primary mechanism for grouping and managing related items - the chapter's `movable` and `moving_ball` tags illustrate both

bulk operations and targeted animation.

Item ids and ordering

Items are assigned integer ids in creation order. Later items are drawn above earlier ones. `raise`/`lower` reorder items; `find` queries items by coordinates or tags; `bbox` returns the bounding box of a set of items. These commands let you build hit-testing, selection, and z-order management without tracking ids manually.

Animation and after

Canvas animation is built on `after`: a procedure moves items a small amount and schedules itself to run again after a short delay. The chapter's `animate` procedure reads the ball's coordinates, bounces it off the walls by negating a velocity delta, and reschedules. `after cancel` stops the loop. For smoother motion, keep the delay short (15-30 ms) and the per-frame movement small. For many simultaneous animations, a single master timer that updates all objects per frame is more efficient than many independent `after` loops.

Export and printing

The canvas can export its contents to a PostScript file with `.c postscript -file name`, as used in Chapter 17's drawing app. For raster formats (PNG, JPEG), capture the window or use the `img::*` extensions. The PostScript export is built in and sufficient for line graphics and diagrams.

What You Learned

- `create line/rectangle/oval/polygon/arc/text/bitmap` draw items; each returns an id.
- Tags group items; `move`, `coords`, and `itemconfigure` accept ids or tags.
- `after ms script` schedules a callback; rescheduling creates an animation loop; `after cancel id` stops it.

Exercises

1. Add a second animated ball with a different color and velocity.
2. Use `itemconfigure` to make the bouncing ball change color each time it hits a wall.
3. Add a button that uses `.c raise`/`.c lower` to change the stacking order of two overlapping shapes.

Chapter 14: Menus, Dialogs, and State

Desktop applications need menu bars, dialog boxes, and a way to manage application state. This chapter covers the `menu` command for menus, the `tk_messageBox`, `tk_getOpenFile`, `tk_getSaveFile`, and `tk_chooseColor` dialogs, and how to wire menu items and keyboard shortcuts to procedures.

Reference: the `menu` man page (menu manual page).

14.1 Menu bars

A menu bar is a `menu` widget attached to the root with `. config -menu .mb`. Cascading menus are added with `add cascade -label ... -menu submenu`. Menu items are added with `add command -label ... -command ...`, `add separator`, `add checkbutton`, and `add radiobutton`. The `-accelerator` option displays a shortcut hint (it does not by itself create the binding - you must add a `bind` for the keystroke). `-underline n` underlines the nth character for keyboard navigation (menu manual page).

14.2 Standard dialogs

Tk provides several built-in dialogs:

- `tk_messageBox -message ... -type yesno -icon question` - a modal message; returns `"yes"`, `"no"`, etc.
- `tk_getOpenFile -filetypes {...}` - a file-open dialog; returns the chosen path or an empty string.
- `tk_getSaveFile` - a file-save dialog.
- `tk_chooseColor -title ...` - a color picker; returns a color name or empty string.

14.3 The tested script

`ch14_menus.tcl` builds a menu bar with File/Edit/Help menus, a text area, and procedures that use the standard dialogs:

```
# ch14_menus.tcl — Dialogs, menus, and application state
# Run with: wish ch14_menus.tcl

package require Tk
wm title . "Menus and Dialogs"
wm geometry . "500x400"

# --- MENU BAR ---
menu .mb -tearoff 0
. config -menu .mb

# File menu
menu .mb.file -tearoff 0
.mb add cascade -label "File" -menu .mb.file -underline 0
.mb.file add command -label "New" -command { newFile } -underline 0 -accelerator "Ctrl+N"
.mb.file add command -label "Open..." -command { openFile } -underline 0 -accelerator "Ctrl+O"
.mb.file add command -label "Save" -command { saveFile } -underline 0 -accelerator "Ctrl+S"
.mb.file add separator
.mb.file add command -label "Exit" -command { exit } -underline 1

# Edit menu
menu .mb.edit -tearoff 0
.mb add cascade -label "Edit" -menu .mb.edit -underline 0
.mb.edit add command -label "Clear" -command { .txt delete 1.0 end }
.mb.edit add separator
.mb.edit add command -label "Select All" -command {
    .txt tag add sel 1.0 end
}

# Help menu
menu .mb.help -tearoff 0
.mb add cascade -label "Help" -menu .mb.help -underline 0
.mb.help add command -label "About" -command {
    tk_messageBox -title "About" -message "Menu Demo\nTcl/Tk 8.6" -type ok
}

# --- TEXT AREA ---
text .txt -width 60 -height 20 -wrap word -font {Courier 10}
pack .txt -fill both -expand 1 -padx 5 -pady 5

# --- DIALOG FUNCTIONS ---
proc newFile {} {
    set answer [tk_messageBox -message "Clear current text?" -type yesno -icon question]
    if {$answer eq "yes"} {
        .txt delete 1.0 end
    }
}

proc openFile {} {
    set filename [tk_getOpenFile -filetypes {
```

```
        {{Text Files} {.txt}}
        {{Tcl Files} {.tcl}}
        {{All Files} *}
    }]
    if {$filename ne ""} {
        set fh [open $filename r]
        .txt delete 1.0 end
        .txt insert end [read $fh]
        close $fh
        wm title . "[file tail $filename]"
    }
}

proc saveFile {} {
    set filename [tk_getSaveFile -filetypes {
        {{Text Files} {.txt}}
        {{All Files} *}
    }]
    if {$filename ne ""} {
        set fh [open $filename w]
        puts $fh [.txt get 1.0 end]
        close $fh
    }
}

# --- BINDINGS ---
bind . <Control-n> { newFile }
bind . <Control-o> { openFile }
bind . <Control-s> { saveFile }

# --- COLOR DIALOG ---
button .color_btn -text "Choose Color" -command {
    set color [tk_chooseColor -title "Pick a color"]
    if {$color ne ""} {
        .txt configure -fg $color
    }
}
pack .color_btn -pady 5

focus .txt
wm protocol . WM_DELETE_WINDOW { exit }
```

14.4 Line-by-line walkthrough

- `menu .mb -tearoff 0` - creates the menu bar with the tear-off feature disabled (`-tearoff 0` removes the dashed tear-off line at the top).
- `. config -menu .mb` - attaches `.mb` as the window's menu bar.
- `menu .mb.file -tearoff 0` and `.mb add cascade -label "File" -menu .mb.file` - creates a submenu and adds it as a cascade.
- `.mb.file add command -label "New" -command { newFile } -accelerator "Ctrl+N" - a menu command with a shortcut hint. The -command` calls the `newFile` procedure.
- `.mb.file add separator` - a visual divider.
- `text .txt -width 60 -height 20 -wrap word` - the main editing area.
- `proc newFile {}` - uses `tk_messageBox -type yesno` to confirm, then clears the text with `.txt delete 1.0 end`. Text indices use a `line.column` format; `1.0` is the start and end is past the last character.

- `proc openFile {}` - calls `tk_getOpenFile` with `-filetypes` to filter the file list; if a file is chosen, it opens it, clears the text, inserts the contents with `.txt insert end [read $fh]`, and updates the window title.
- `proc saveFile {}` - calls `tk_getSaveFile`, then writes the text widget's contents (`puts $fh [.txt get 1.0 end]`) to the chosen file.
- `bind . <Control-n> { newFile }` etc. - the keyboard shortcuts that the menu hints at; without these, the accelerators are just labels.
- The color button uses `tk_chooseColor`; if a color is chosen, it sets the text widget's foreground.

> **Tip:** The `-accelerator` option only *displays* the shortcut text. You
> must also add a matching `bind` for the keystroke, or the shortcut will not
> actually work.

Concepts in Depth

Menu structure

A menu bar is a `menu` widget attached via `. config -menu .mb`. Each top-level entry is usually a cascade (`add cascade`) pointing at a submenu created with `menu .mb.sub`. Submenus contain `command`, `checkbutton`, `radiobutton`, and `separator` entries. Cascades can be nested arbitrarily deep, though one level (a menu bar of cascades) is the norm. The `-tearoff 0` option removes the dashed tear-off handle that older Tk shows by default.

Accelerators vs. bindings

An `-accelerator` string is purely visual: it displays "Ctrl+S" next to the menu item but does not create the shortcut. You must add a matching `bind` yourself, and that binding should call the same command the menu item calls. Centralizing the command in a procedure (as the chapter does with `saveFile`) keeps the menu item and the keyboard shortcut in sync and makes the action testable independently of the UI.

Standard dialogs and modality

The `tk_messageBox`, `tk_getOpenFile`, `tk_getSaveFile`, and `tk_chooseColor` commands are implemented in Tcl itself and wrap platform-native dialogs where possible. They are *modal*: they block until the user responds, returning the choice as a string. For a non-modal, custom dialog, create a `toplevel` window and use `tkwait window` or `vwait` to wait for it. The built-in dialogs cover the vast majority of needs and should be preferred over hand-rolled equivalents for consistency and accessibility.

Application state and the dirty flag

The notepad project in Chapter 16 generalizes the pattern shown here: a global flag tracks whether the document has unsaved changes, and every action that could lose data (New, Open, Exit) consults it and prompts to save. Keeping this flag accurate is the job of the <<Modified>> virtual event on the text widget. This state-management discipline is what separates a toy editor from a reliable one.

Virtual events

<<Cut>>, <<Copy>>, <<Paste>>, <<Modified>>, and <<Clear>> are *virtual events* - named events that Tk generates or that you can generate with `event generate`. They let you trigger a widget's built-in behavior (such as the text widget's clipboard operations) from a menu item without reimplementing the operation. Chapter 16 uses this to wire the Edit menu to the text widget.

What You Learned

- `menu` and `add cascade`/`add command`/`add separator` build menu bars.
- `. config -menu .mb` attaches a menu bar to the window.
- `tk_messageBox`, `tk_getOpenFile`, `tk_getSaveFile`, and `tk_chooseColor` provide standard dialogs.
- `-accelerator` is a label only; add a `bind` to make the shortcut work.
- Text widget indices are `line.column`; `1.0` is the start, `end` is the end.

Exercises

1. Add an Edit menu item "Insert Date" that inserts the current date (from `clock format [clock seconds]`) at the cursor.
2. Add a Format menu with a checkbutton that toggles word wrap on and off.
3. Make the File > Open menu item also update the status bar with the file name.

Chapter 15: Project - Calculator

This chapter brings together variables, procedures, the `grid` geometry manager, widgets, and bindings to build a working calculator. The project demonstrates a data-driven button layout, state management with global variables, keyboard input, and error handling for division by zero.

15.1 Design overview

The calculator keeps three pieces of state in global variables:

- `display` - the string currently shown on the screen.
- `stored` - the first operand, saved when an operator is pressed.
- `operation` - the pending operator (`+`, `-`, `*`, `/`), or `""` (none), or `"just"` (a sentinel meaning "the last action was =, so the next digit starts a fresh number").

The buttons are generated from a `layout` list of `{label type}` pairs, which keeps the code compact and easy to modify.

15.2 The tested script

`ch15_calculator.tcl`:

```
# ch15_calculator.tcl — Project: Calculator
# Run with: wish ch15_calculator.tcl

package require Tk
wm title . "Calculator"
wm geometry . "250x300"

# Variables
set display "0"
set stored 0
set operation ""

# Display label
label .display -textvariable display -font {Courier 20} -anchor e -relief sunken -bd 2
grid .display - - - - -sticky ew -padx 5 -pady 5

# Button layout: {label command_type}
set layout {
    {7 digit} {8 digit} {9 digit} {/ op} {C clear}
    {4 digit} {5 digit} {6 digit} {* op} {"" spacer}
    {1 digit} {2 digit} {3 digit} {- op} {= equals}
    {0 digit} {. digit} {+ op} {"" spacer} {"" spacer}
}

set row 1
set col 0
foreach {label_text cmd_type} $layout {
    if {$cmd_type eq "spacer"} {
        # Empty cell - skip
    } elseif {$cmd_type eq "clear"} {
        button .b_${row}_${col} -text $label_text -width 3 -command {
            set display "0"
            set stored 0
            set operation ""
        }
        grid .b_${row}_${col} -row $row -column $col -sticky nsew -padx 2 -pady 2
    } elseif {$cmd_type eq "equals"} {
        button .b_${row}_${col} -text $label_text -width 3 -command { doCalculate }
        grid .b_${row}_${col} -row $row -column $col -sticky nsew -padx 2 -pady 2
    } elseif {$cmd_type eq "digit"} {
        button .b_${row}_${col} -text $label_text -width 3 \
            -command [list digitPressed $label_text]
        grid .b_${row}_${col} -row $row -column $col -sticky nsew -padx 2 -pady 2
    } elseif {$cmd_type eq "op"} {
        button .b_${row}_${col} -text $label_text -width 3 \
            -command [list opPressed $label_text]
        grid .b_${row}_${col} -row $row -column $col -sticky nsew -padx 2 -pady 2
    }
    incr col
    if {$col > 4} { set col 0; incr row }
```

```
}

proc digitPressed {d} {
    global display stored operation
    if {$display eq "0" || $operation eq "just"} {
        set display $d
        set operation ""
    } else {
        set display $display$d
    }
}

proc opPressed {op} {
    global display stored operation
    set stored $display
    set operation $op
    set display "0"
}

proc doCalculate {} {
    global display stored operation
    if {$operation eq "" || $operation eq "just"} return
    set result 0
    switch -- $operation {
        "+" { set result [expr {$stored + $display}] }
        "-" { set result [expr {$stored - $display}] }
        "*" { set result [expr {$stored * $display}] }
        "/" {
            if {$display == 0} {
                tk_messageBox -message "Cannot divide by zero"
                set display "0"
                set stored 0
                set operation ""
                return
            }
            set result [expr {$stored / $display}]
        }
    }
    set display $result
    set stored 0
    set operation "just"
}

# Make grid cells expand
for {set i 0} {$i < 5} {incr i} {
    grid columnconfigure . $i -weight 1
}
for {set i 1} {$i < 5} {incr i} {
    grid rowconfigure . $i -weight 1
}
```

```
# Keyboard bindings
bind . <Key> {
    set k %A
    if {[string is digit $k]} {
        digitPressed $k
    } elseif {$k eq "."} {
        digitPressed $k
    } elseif {$k eq "+" || $k eq "-" || $k eq "*" || $k eq "/"} {
        opPressed $k
    }
}
bind . <Return> { doCalculate }
bind . <Escape> { set display "0"; set stored 0; set operation "" }

wm protocol . WM_DELETE_WINDOW { exit }
```

15.3 Line-by-line walkthrough

- `label .display -textvariable display -font {Courier 20} -anchor e -relief sunken - the display is a label bound to display`, right-aligned, with a sunken border so it looks like a screen.
- `grid .display - - - -` the display spans four columns (three - placeholders) in row 0.
- The `layout` list describes every button: its label and a type (`digit`, `op`, `clear`, `equals`, `spacer`). A `foreach {label_text cmd_type} $layout` loop generates the buttons.
- For a `digit` button, `-command [list digitPressed $label_text]` uses `list` to build the command script safely, so `digitPressed` receives the correct label even though the loop variable changes.
- `proc digitPressed {d}` - if the display is `"0"` or the last action was `=` (the `just` sentinel), it replaces the display with the new digit; otherwise it appends the digit. This gives natural calculator behavior.
- `proc opPressed {op}` - saves the current display into `stored`, records the operator, and resets `display` to `"0"` ready for the second operand.
- `proc doCalculate {}` - a `switch` on the operator computes the result with `expr`. The `/` branch guards against division by zero with a message box and a reset. After computing, it sets `display` to the result, clears `stored`, and sets `operation` to `"just"` so the next digit starts fresh.
- The `grid columnconfigure/rowconfigure` loops with `-weight 1` make all cells expand equally so the buttons resize with the window.
- `bind . <Key> { set k %A; ... }` - the `%A` substitution gives the character typed; the script routes digits, `.`, and operators to the right procedures, giving full keyboard support. `<Return>` triggers calculation and `<Escape>` resets.

> **Tip:** Using `[list digitPressed $label_text]` instead of a literal string like `"digitPressed $label_text"` is the safe way to build a callback that depends on a loop variable. `list` produces a properly quoted command, so labels with spaces or special characters would still work.

Concepts in Depth

State machines in the small

A calculator is a tiny state machine with states like "entering first operand", "operator pressed, entering second operand", and "result shown". The three

globals (`display`, `stored`, `operation`) encode this state, and the `"just"` sentinel marks the "result shown" state so the next digit starts fresh rather than appending to the result. Recognizing this structure makes the logic easy to extend (adding a percent or sign button is just another transition) and keeps edge cases - like what happens when the user types digits after = - well-defined.

Data-driven UI generation

The `layout` list pairs each button's label with a type. The `foreach` loop generates a button for each entry, choosing the command based on the type. This pattern scales: to add a `+/-` button, append one line to `layout` and add one branch to the loop. Contrast this with writing each button by hand, which would duplicate the `grid` and option boilerplate eighteen times. Data-driven generation is a hallmark of idiomatic Tcl/Tk and recurs in the later projects (the board loop in Chapter 18 generates nine buttons from a single loop).

Safe callbacks with list

When a callback must depend on a loop variable, `[list procName $arg]` produces a properly quoted command string. Plain string interpolation (`"procName $label_text"`) would break if the label contained spaces or special characters. Because calculator buttons here are simple digits and operators, the danger is small, but the habit pays off immediately in Chapter 18, where the cell index is the argument. Using `list` to build callbacks is one of the most important Tcl safety patterns.

Resizing with grid weights

The `for` loops over `grid columnconfigure` and `grid rowconfigure` set `-weight 1` on every cell, so when the window grows, every cell absorbs an equal share of the extra space, and the buttons (with `-sticky nsew`) fill their cells. Without these weights, the buttons would stay their default size and cluster in the corner. Weights are the single most important `grid` skill for resize-friendly layouts.

Keyboard input via %A

The <Key> binding uses %A to get the typed character, then routes it with `string is digit` and a chain of `elseif` checks. This gives the calculator a full keyboard interface with very little code. Note that %A is empty for some keys (like Shift), so guarding with `string is` prevents mis-routing.

What You Learned

- Data-driven UI generation with a `layout` list keeps button code compact.
- Global variables (`display`, `stored`, `operation`) track calculator state.
- A "`just`" sentinel cleanly separates "result of =" from "new input".
- `grid columnconfigure/rowconfigure` with `-weight 1` make a resizing grid.
- `[list procName $arg]` builds safe per-button callbacks.
- Key bindings with `%A` enable keyboard input.

Exercises

1. Add a `+/-` button that negates the current display.
2. Add a `%` button that converts the current display to a percentage of `stored`.
3. Add a memory feature (M+, MR, MC) using a fourth global variable.

Chapter 16: Project - Notepad

This project builds a multi-feature text editor with menus, a scrollbar, a status bar, change tracking, and "save changes?" prompts. It exercises the `text` widget, the `scrollbar` widget, file I/O, dialogs, and the `<<Modified>>` virtual event.

16.1 Design overview

The notepad tracks two globals: `current_file` (the path of the open file, or "") and `text_changed` (a dirty flag). The text widget is linked to a vertical scrollbar, and a status label at the bottom reports state. File menus (New, Open, Save, Save As) each call a procedure that manages the dirty flag and the window title. The `<<Modified>>` binding on the text widget updates the dirty flag automatically whenever the user edits text.

16.2 The tested script

`ch16_notepad.tcl`:

```
# ch16_notepad.tcl — Project: Notes/Text Editor
# Run with: wish ch16_notepad.tcl

package require Tk
wm title . "Untitled - Notepad"
wm geometry . "600x400"

# Variables
set current_file ""
set text_changed 0

# --- MENU BAR ---
menu .mb -tearoff 0
. config -menu .mb

# File menu
menu .mb.file -tearoff 0
.mb add cascade -label "File" -menu .mb.file -underline 0
.mb.file add command -label "New" -command newFile -accelerator "Ctrl+N" -underline 0
.mb.file add command -label "Open..." -command openFile -accelerator "Ctrl+O" -underline 0
.mb.file add command -label "Save" -command saveFile -accelerator "Ctrl+S" -underline 0
.mb.file add command -label "Save As..." -command saveAsFile -underline 1
.mb.file add separator
.mb.file add command -label "Exit" -command exit -underline 1

# Edit menu
menu .mb.edit -tearoff 0
.mb add cascade -label "Edit" -menu .mb.edit -underline 0
.mb.edit add command -label "Cut" -command { event generate .txt <<Cut>> } -accelerator "Ctrl+X"
.mb.edit add command -label "Copy" -command { event generate .txt <<Copy>> } -accelerator "Ctrl+C"
.mb.edit add command -label "Paste" -command { event generate .txt <<Paste>> } -accelerator "Ctrl+V"
.mb.edit add separator
.mb.edit add command -label "Select All" -command { .txt tag add sel 1.0 end } -accelerator "Ctrl+A"
.mb.edit add command -label "Clear" -command { .txt delete 1.0 end }

# Format menu
menu .mb.format -tearoff 0
.mb add cascade -label "Format" -menu .mb.format -underline 0
.mb.format add command -label "Word Wrap On" -command { .txt configure -wrap word }
.mb.format add command -label "Word Wrap Off" -command { .txt configure -wrap none }
.mb.format add separator
.mb.format add command -label "Font Size +" -command {
    set fs [.txt cget -font]
    set sz [lindex $fs 1]
    .txt configure -font [lreplace $fs 1 1 [expr {$sz + 1}]]
}
.mb.format add command -label "Font Size -" -command {
    set fs [.txt cget -font]
    set sz [lindex $fs 1]
    if {$sz > 6} {
```

```
        .txt configure -font [lreplace $fs 1 1 [expr {$sz - 1}]]
    }
}

# Help menu
menu .mb.help -tearoff 0
.mb add cascade -label "Help" -menu .mb.help -underline 0
.mb.help add command -label "About" -command {
    tk_messageBox -title "About" -message "Notepad\nA Tcl/Tk text editor\nTcl/Tk 8.6" -type ok
}

# --- TEXT WIDGET WITH SCROLLBAR ---
frame .tf
text .txt -yscrollcommand ".sb set" -wrap word -font {Courier 11} -undo 1
scrollbar .sb -command ".txt yview"
pack .txt -side left -fill both -expand 1
pack .sb -side right -fill y
pack .tf -fill both -expand 1

# Status bar
label .status -text "Ready" -anchor w -relief sunken -bd 1
pack .status -fill x

# --- FUNCTIONS ---
proc newFile {} {
    global current_file text_changed
    if {$text_changed} {
        set answer [tk_messageBox -message "Save changes?" -type yesnocancel -icon question]
        if {$answer eq "yes"} { saveFile }
        if {$answer eq "cancel"} return
    }
    .txt delete 1.0 end
    set current_file ""
    set text_changed 0
    wm title . "Untitled - Notepad"
    .status configure -text "New file"
}

proc openFile {} {
    global current_file text_changed
    if {$text_changed} {
        set answer [tk_messageBox -message "Save changes?" -type yesnocancel -icon question]
        if {$answer eq "yes"} { saveFile }
        if {$answer eq "cancel"} return
    }
    set filename [tk_getOpenFile -filetypes {
        {{Text Files} {.txt}}
        {{Tcl Files} {.tcl}}
        {{All Files} *}
    }]
```

```
    if {$filename eq ""} return
    set fh [open $filename r]
    .txt delete 1.0 end
    .txt insert end [read $fh]
    close $fh
    set current_file $filename
    set text_changed 0
    wm title . "[file tail $filename] - Notepad"
    .status configure -text "Opened: $filename"
}

proc saveFile {} {
    global current_file text_changed
    if {$current_file eq ""} {
        saveAsFile
        return
    }
    set fh [open $current_file w]
    puts -nonewline $fh [.txt get 1.0 "end-1c"]
    close $fh
    set text_changed 0
    .status configure -text "Saved: $current_file"
}

proc saveAsFile {} {
    global current_file text_changed
    set filename [tk_getSaveFile -filetypes {
        {{Text Files} {.txt}}
        {{All Files} *}
    }]
    if {$filename eq ""} return
    set fh [open $filename w]
    puts -nonewline $fh [.txt get 1.0 "end-1c"]
    close $fh
    set current_file $filename
    set text_changed 0
    wm title . "[file tail $filename] - Notepad"
    .status configure -text "Saved: $filename"
}

# Track changes
bind .txt <<Modified>> {
    if {[%W edit modified]} {
        set text_changed 1
        .status configure -text "Modified"
        %W edit modified 0
    }
}

# Keyboard shortcuts
```

```
bind . <Control-n> { newFile }
bind . <Control-o> { openFile }
bind . <Control-s> { saveFile }
bind . <Control-a> { .txt tag add sel 1.0 end; break }

focus .txt
wm protocol . WM_DELETE_WINDOW {
    if {$text_changed} {
        set answer [tk_messageBox -message "Save changes before closing?" -type yesnocancel -icon question]
        if {$answer eq "yes"} { saveFile }
        if {$answer eq "cancel"} return
    }
    exit
}
```

16.3 Line-by-line walkthrough

- `text .txt -yscrollcommand ".sb set" -wrap word -undo 1` - the text widget;

`-yscrollcommand ".sb set"` keeps the linked scrollbar in sync as the text scrolls, and `-undo 1` enables built-in undo/redo.
- `scrollbar .sb -command ".txt yview"` - the scrollbar; its `-command` tells the text widget to scroll when the scrollbar is moved. This two-way link (text updates scrollbar, scrollbar updates text) is the standard pattern.
- `pack .txt -side left -fill both -expand 1` and `pack .sb -side right -fill y` - pack the text on the left expanding, and the scrollbar on the right.
- The File menu's New/Open/Save/Save As commands call procedures.
- `proc newFile {}` - if the text is dirty, it asks `tk_messageBox -type yesnocancel`; "yes" saves, "cancel" aborts. Otherwise it clears the text, resets `current_file` and the dirty flag, and updates the title.
- `proc openFile {}` - same dirty check, then `tk_getOpenFile`, then reads the file into the text widget with `.txt insert end [read $fh]`.
- `proc saveFile {}` - if there is no `current_file`, it delegates to `saveAsFile`; otherwise it writes the text to `current_file`. Note `puts -nonewline $fh [.txt get 1.0 "end-1c"]` - the index `"end-1c"` strips the trailing newline that `get 1.0 end` would otherwise include, so saving and reloading is round-trip stable.
- `proc saveAsFile {}` - uses `tk_getSaveFile` and writes the text.
- `bind .txt <<Modified>> { if {[%W edit modified]} { set text_changed 1; ...; %W edit modified 0 } }` - the `<<Modified>>` virtual event fires when the text widget's modified flag changes. `%W` is the widget that received the event. The handler sets `text_changed`, updates the status, and resets the modified flag so the event can fire again on the next change.
- Keyboard shortcuts (`<Control-n>` etc.) mirror the menu accelerators.
- The `WM_DELETE_WINDOW` handler asks to save before quitting, so closing the window does not lose unsaved work.

> **Tip:** The `<<Cut>>`, `<<Copy>>`, `<<Paste>>`, and `<<Modified>>` names are
> **virtual events**. Tk generates them automatically; you can also generate
> them yourself with `event generate .txt <<Cut>>`, which is how the Edit menu
> triggers clipboard operations on the text widget.

Concepts in Depth

The text/scrollbar link

The canonical pattern for a scrolling text widget is two options: `-yscrollcommand {.sb set}` on the `text` and `-command {.txt yview}` on the `scrollbar`. The text tells the scrollbar how much is visible; the scrollbar

tells the text where to scroll. This bidirectional link is the same for any scrollable widget (`listbox`, `canvas`, `ttk::treeview`). Getting both options right is essential; omitting either half leaves the widget or the scrollbar unresponsive.

Indices: line.column, word, and end-N

Text widget indices are strings. `1.0` is line 1, column 0 (the start). `end` is just past the last character. `"end-1c"` is one character before `end`, which strips the trailing newline that `get 1.0 end` includes - the reason `saveFile` uses `"end-1c"` for a round-trip-stable save. Other index forms include `wordstart`, `wordend`, `linestart`, `lineend`, and named marks. The `search` subcommand finds text and returns an index, which is the basis for a Find feature (Exercise 1).

The dirty flag and <<Modified>>

The text widget maintains an internal "modified" flag. Toggling it generates the `<<Modified>>` virtual event. The notepad's binding checks `[%W edit modified]`, sets the global dirty flag, updates the status bar, and resets the modified flag (`%W edit modified 0`) so the event can fire again on the next change. This is the cleanest way to track edits without polling. `%W` is the widget that received the event, making the binding reusable across multiple text widgets.

Save-changes prompts

Every action that would discard unsaved work (New, Open, the window close) calls a shared check that uses `tk_messageBox -type yesnocancel`. "yes" saves, "no" discards, "cancel" aborts the action. Centralizing this check in a procedure (or at least a consistent pattern) prevents the classic bug where one path forgets to prompt and silently loses data. The window-close handler is especially important: without `wm protocol . WM_DELETE_WINDOW`, closing the window would bypass the prompt entirely.

Undo/redo and event generation

`-undo 1` enables the text widget's built-in undo/redo stack, accessible through the `edit` subcommands (`edit undo`, `edit redo`, `edit canundo`, `edit modified`, `edit separator`). The Edit menu uses `event generate .txt <<Cut>>` (and `<<Copy>>`, `<<Paste>>`) to drive the text widget's native clipboard operations rather than reimplementing them. This keeps behavior consistent with the platform and reduces code.

What You Learned

- Linking `text` and `scrollbar` with `-yscrollcommand`/`-command` gives a scrolling editor.
- `<<Modified>>` is a virtual event for change tracking; `%W` is the source widget.
- `tk_messageBox -type yesnocancel` drives "save changes?" prompts.
- `"end-1c"` index trims the trailing newline for stable round-trips.
- `wm protocol . WM_DELETE_WINDOW` lets you intercept the window close.

Exercises

1. Add a Find feature: an entry and a button that highlights the next occurrence of a search string using `.txt search`.
2. Add a status bar item showing the current line and column using `index insert`.
3. Add a "Recent Files" submenu that remembers the last few opened files.

Chapter 17: Project - Drawing Application

This project builds a drawing program with a tool palette (pen, line, rectangle, oval, eraser), a color picker, a clear command, and PostScript export. It combines the canvas, radio buttons, `tk_chooseColor`, bindings for mouse drag drawing, and `trace` to keep a status bar in sync.

17.1 Design overview

Five tools are selected by radio buttons bound to the `tool` variable. Mouse events drive drawing:

- `<Button-1>` starts a stroke.
- `<B1-Motion>` continues a freehand stroke (pen/eraser).
- `<ButtonRelease-1>` completes a shape (line/rect/oval) drawn from the press point to the release point.

Two `trace add variable ... write` callbacks update the status bar whenever `tool` or `color` changes, demonstrating data-driven UI updates.

17.2 The tested script

`ch17_draw.tcl`:

```
# ch17_draw.tcl — Project: Drawing Application
# Run with: wish ch17_draw.tcl

package require Tk
wm title . "Drawing App"
wm geometry . "700x500"

# --- VARIABLES ---
set tool "pen"
set color "black"
set last_x 0
set last_y 0
set item_id 0

# --- MENU BAR ---
menu .mb -tearoff 0
. config -menu .mb

menu .mb.file -tearoff 0
.mb add cascade -label "File" -menu .mb.file
.mb.file add command -label "Clear Canvas" -command { .c delete all; set item_id 0 }
.mb.file add command -label "Save as PostScript" -command {
    set filename [tk_getSaveFile -filetypes {{{PostScript} {.ps}} {{"All Files"} *}}]
    if {$filename ne ""} {
        .c postscript -file $filename
        tk_messageBox -message "Saved to $filename"
    }
}
.mb.file add separator
.mb.file add command -label "Exit" -command exit

# --- TOOLBAR ---
frame .toolbar -relief raised -bd 2

# Tool buttons
label .toolbar.tool_lbl -text "Tool:"
set tools {pen line rect oval eraser}
set tool_labels {Pen Line Rect Oval Eraser}
foreach t $tools tl $tool_labels {
    radiobutton .toolbar.t_$t -text $tl -variable tool -value $t
    pack .toolbar.t_$t -side left -padx 2
}

# Separator
frame .toolbar.sep1 -width 2 -relief sunken -bd 1
pack .toolbar.sep1 -side left -fill y -padx 5

# Color button
label .toolbar.color_lbl -text "Color:"
button .toolbar.color_btn -textvariable color -command {
```

```
    set c [tk_chooseColor -title "Choose Color"]
    if {$c ne ""} { set color $c }
}
pack .toolbar.color_lbl .toolbar.color_btn -side left -padx 2

# Separator
frame .toolbar.sep2 -width 2 -relief sunken -bd 1
pack .toolbar.sep2 -side left -fill y -padx 5

# Clear button
button .toolbar.clear -text "Clear" -command { .c delete all; set item_id 0 }
pack .toolbar.clear -side left -padx 2

pack .toolbar -fill x

# --- CANVAS ---
canvas .c -width 680 -height 420 -bg white -relief sunken -bd 2
pack .c -fill both -expand 1 -padx 5 -pady 5

# --- DRAWING LOGIC ---
bind .c <Button-1> {
    set last_x %x
    set last_y %y
    if {$tool eq "pen"} {
        set item_id [.c create line $last_x $last_y $last_x $last_y -fill $color -width 2 -smooth 1]
    } elseif {$tool eq "eraser"} {
        set item_id [.c create line $last_x $last_y $last_x $last_y -fill white -width 15]
    } elseif {$tool eq "line"} {
        # Will draw on release
    } elseif {$tool eq "rect"} {
        # Will draw on release
    } elseif {$tool eq "oval"} {
        # Will draw on release
    }
}

bind .c <B1-Motion> {
    set x %x
    set y %y
    if {$tool eq "pen"} {
        .c coords $item_id [concat [.c coords $item_id] $x $y]
    } elseif {$tool eq "eraser"} {
        .c coords $item_id [concat [.c coords $item_id] $x $y]
    }
    set last_x $x
    set last_y $y
}

bind .c <ButtonRelease-1> {
    set x %x
```

```
    set y %y
    if {$tool eq "line"} {
        .c create line $last_x $last_y $x $y -fill $color -width 2
    } elseif {$tool eq "rect"} {
        .c create rectangle $last_x $last_y $x $y -outline $color -width 2
    } elseif {$tool eq "oval"} {
        .c create oval $last_x $last_y $x $y -outline $color -width 2
    }
}

# --- STATUS BAR ---
label .status -text "Tool: $tool | Color: $color | Right-click to change tool" -anchor w -relief sunken -bd 1
pack .status -fill x

# Update status on tool change
trace add variable tool write {apply {{args} {
    global tool color
    .status configure -text "Tool: $tool | Color: $color"
}}}

trace add variable color write {apply {{args} {
    global tool color
    .status configure -text "Tool: $tool | Color: $color"
}}}

wm protocol . WM_DELETE_WINDOW { exit }
```

17.3 Line-by-line walkthrough

- The menu bar's "Save as PostScript" item uses `.c postscript -file $filename` - the canvas can export its contents to a PostScript file, a handy built-in feature.
- `foreach t $tools tl $tool_labels { radiobutton .toolbar.t_$t -text $tl -variable tool -value $t }` - generates the tool radio buttons in parallel from two lists.
- `button .toolbar.color_btn -textvariable color -command { ... tk_chooseColor ... }` - the color button shows the current color and opens the color picker.
- The canvas bindings:
- On <Button-1>, `last_x`/`last_y` record the start point. For the pen or eraser, a `create line` item is started (a zero-length line that will be extended). For line/rect/oval, nothing is drawn yet - the shape appears on release.
- On <B1-Motion> (mouse moved with button held), the pen and eraser extend their line item by appending coordinates with `.c coords $item_id [concat [.c coords $item_id] $x $y]`.
- On <ButtonRelease-1>, the line/rect/oval tools create the final shape from `last_x`/`last_y` to the release point.
- The status label is created with an initial string, then two `trace add variable ... write callbacks update it whenever tool or color` changes.

The trace body uses `apply` to define an anonymous procedure inline, avoiding the need for a named procedure just for the trace.

> **Tip:** `trace add variable name write callback` runs `callback` whenever `name` is written. This is the data-binding mechanism: change the variable (e.g., via a radio button) and the UI updates automatically, with no extra code in the button command.

> **Note:** The freehand pen uses a single `create line` item extended with `-smooth 1` for a smooth curve. Appending coordinates to an existing line item (rather than creating many small segments) is efficient and keeps the stroke as one selectable object.

Concepts in Depth

Drawing models: freehand vs. shape

The app supports two interaction models. Freehand tools (pen, eraser) create a single `line` item on `Button-1` and extend it on every `B1-Motion` by appending coordinates with `.c coords $id [concat [.c coords $id] $x $y]`. Shape tools (line, rect, oval) record the start point on press and create the final item on release, using the press and release coordinates as corners. This split keeps the binding logic simple while supporting very different drawing styles.

Extending a line item in place

Appending to an existing line item (rather than creating many short segments) is both efficient and semantically correct: the whole stroke remains one selectable, movable object. The `coords` command with a list of points replaces the item's coordinate list entirely, so `concat` merges the old points with the new ones. The `-smooth 1` option on the pen line turns the polyline into a smoothed curve for a natural feel.

trace as data binding

`trace add variable name write callback` arranges for `callback` to run whenever `name` is written. The drawing app uses two traces - on `tool` and `color` - to update the status bar automatically, so the radio buttons and color picker need no explicit UI-update code. This is Tk's general data-binding mechanism: change the variable, the UI follows. The `apply` form used for the callback bodies creates an anonymous procedure inline, avoiding the need for a named procedure just to serve the trace. Traces also fire on `unset` and (with `array`) on element changes.

Toolbars and frames

The toolbar is a `frame` with `-relief raised` containing radio buttons, labels, buttons, and thin separator frames. Packing them `-side left` lays them out horizontally. Using a frame as a container lets the toolbar be packed as a unit below the menu bar, separate from the canvas's layout - again the "one geometry manager per parent" pattern, with the frame as the boundary.

PostScript export

`.c postscript -file $filename` writes the canvas's vector contents to a PostScript file, suitable for printing or converting to PDF. Options control page size, color, and rotation. Because the export is vector-based, it scales without loss. For raster export, you would use the `img::*` extensions or a screenshot. The built-in PostScript export is enough for line drawings and diagrams, which is what a drawing app produces.

What You Learned

- Canvas mouse bindings (`<Button-1>`, `<B1-Motion>`, `<ButtonRelease-1>`) drive freehand and shape drawing.
- `.c coords itemId [concat [.c coords itemId] $x $y]` extends a line item.
- `trace add variable name write callback` updates the UI when data changes.
- `.c postscript -file $filename` exports the canvas to PostScript.
- `tk_chooseColor` provides a color picker.

Exercises

1. Add a "fill" checkbox; when on, rectangles and ovals are filled with the current color instead of outlined.
2. Add a line-width spinbox that sets the `-width` of new items.
3. Add an "undo" button that deletes the most recently created item using `.c find closest` or a stored id stack.

Chapter 18: Project - Tic-Tac-Toe Game

The final project is a complete tic-tac-toe game with a scoreboard, win detection, draw detection, score tracking, and a reset feature. It combines procedures, lists, the `grid` manager, widgets, and game logic.

18.1 Design overview

The board is stored as a Tcl list of nine strings (initially all `""`). Players alternate between `"X"` and `"O"`. A `checkWin` procedure tests all eight winning lines. The scoreboard uses an array (`scores(X)`, `scores(O)`, `scores(draw)`) bound to labels with `-textvariable`. A "New Game" button resets the board while keeping the scores.

18.2 The tested script

`ch18_game.tcl`:

```
# ch18_game.tcl — Project: Tic-Tac-Toe Game
# Run with: wish ch18_game.tcl

package require Tk
wm title . "Tic-Tac-Toe"
wm geometry . "320x400"

# Game state
set board {"" "" "" "" "" "" "" "" ""}
set current_player "X"
set game_over 0
set moves 0
set scores(X) 0
set scores(O) 0
set scores(draw) 0

# --- SCOREBOARD ---
frame .scoreboard
label .scoreboard.x_lbl -text "X: " -font {Helvetica 12 bold}
label .scoreboard.x_val -textvariable scores(X) -font {Helvetica 12} -fg blue
label .scoreboard.o_lbl -text "  O: " -font {Helvetica 12 bold}
label .scoreboard.o_val -textvariable scores(O) -font {Helvetica 12} -fg red
label .scoreboard.d_lbl -text "  Draws: " -font {Helvetica 12 bold}
label .scoreboard.d_val -textvariable scores(draw) -font {Helvetica 12} -fg gray
pack .scoreboard.x_lbl .scoreboard.x_val .scoreboard.o_lbl .scoreboard.o_val .scoreboard.d_lbl .scoreboard.d_val -side left
pack .scoreboard -pady 5

# --- STATUS ---
label .status -text "Player X's turn" -font {Helvetica 14 bold} -fg blue
pack .status -pady 5

# --- GAME BOARD ---
frame .board
for {set i 0} {$i < 9} {incr i} {
    set row [expr {$i / 3}]
    set col [expr {$i % 3}]
    button .board.b$i -text "" -font {Helvetica 24 bold} -width 4 -height 2 \
        -command [list cellClick $i]
    grid .board.b$i -row $row -column $col -sticky nsew -padx 2 -pady 2
}
for {set i 0} {$i < 3} {incr i} {
    grid columnconfigure .board $i -weight 1
    grid rowconfigure .board $i -weight 1
}
pack .board -padx 10 -pady 10

# --- CONTROL BUTTONS ---
frame .controls
button .controls.new -text "New Game" -command newGame
button .controls.reset -text "Reset Scores" -command resetScores
```

```
pack .controls.new .controls.reset -side left -padx 10
pack .controls -pady 5

# --- LOGIC ---
proc cellClick {idx} {
    global board current_player game_over moves
    if {$game_over} return
    if {[lindex $board $idx] ne ""} return

    set board [lreplace $board $idx $idx $current_player]
    .board.b$idx configure -text $current_player -state disabled
    if {$current_player eq "X"} {
        .board.b$idx configure -fg blue
    } else {
        .board.b$idx configure -fg red
    }
    incr moves

    # Check for win
    set winner [checkWin]
    if {$winner ne ""} {
        set game_over 1
        .status configure -text "Player $winner wins!" -fg green
        incr ::scores($winner)
        highlightWin $winner
        return
    }

    # Check for draw
    if {$moves >= 9} {
        set game_over 1
        .status configure -text "It's a draw!" -fg gray
        incr ::scores(draw)
        return
    }

    # Switch player
    if {$current_player eq "X"} {
        set current_player "O"
        .status configure -text "Player O's turn" -fg red
    } else {
        set current_player "X"
        .status configure -text "Player X's turn" -fg blue
    }
}

proc checkWin {} {
    global board
    set wins {
        {0 1 2} {3 4 5} {6 7 8}
```

```
        {0 3 6} {1 4 7} {2 5 8}
        {0 4 8} {2 4 6}
    }
    foreach line $wins {
        set a [lindex $board [lindex $line 0]]
        set b [lindex $board [lindex $line 1]]
        set c [lindex $board [lindex $line 2]]
        if {$a ne "" && $a eq $b && $a eq $c} {
            return $a
        }
    }
    return ""
}

proc highlightWin {winner} {
    global board
    set wins {
        {0 1 2} {3 4 5} {6 7 8}
        {0 3 6} {1 4 7} {2 5 8}
        {0 4 8} {2 4 6}
    }
    foreach line $wins {
        set a [lindex $board [lindex $line 0]]
        set b [lindex $board [lindex $line 1]]
        set c [lindex $board [lindex $line 2]]
        if {$a ne "" && $a eq $b && $a eq $c} {
            foreach idx $line {
                .board.b$idx configure -bg lightgreen
            }
        }
    }
    # Disable remaining cells
    for {set i 0} {$i < 9} {incr i} {
        .board.b$i configure -state disabled
    }
}

proc newGame {} {
    global board current_player game_over moves
    set board {"" "" "" "" "" "" "" "" ""}
    set current_player "X"
    set game_over 0
    set moves 0
    .status configure -text "Player X's turn" -fg blue
    for {set i 0} {$i < 9} {incr i} {
        .board.b$i configure -text "" -state normal -bg [lindex [.board.b$i configure -bg] 3]
    }
}

proc resetScores {} {
```

```
    global scores
    set scores(X) 0
    set scores(O) 0
    set scores(draw) 0
}

wm protocol . WM_DELETE_WINDOW { exit }
```

18.3 Line-by-line walkthrough

- `set board {"" "" "" "" "" "" "" "" ""}` - a nine-element list representing the cells, all empty.

- `set current_player "X"`, `set game_over 0`, `set moves 0` - game state.

- The scoreboard labels use `-textvariable scores(X)` etc., so the displayed

numbers update automatically when the array elements change.
- The board buttons are generated in a loop: `button .board.b$i ... -command [list cellClick $i], then grid .board.b$i -row $row -column $col`. The row and column are computed with integer division and modulo. `grid columnconfigure/rowconfigure with -weight 1` make the buttons expand into a square grid.
- `proc cellClick {idx}` - the heart of the game:
- It returns immediately if the game is over or the cell is taken (`[lindex $board $idx] ne ""`).
- It updates the board list with `lreplace` (`set board [lreplace $board $idx $idx $current_player]`), sets the button text, disables the button, and colors it blue (X) or red (O).
- It calls `checkWin`; if there is a winner, it sets `game_over`, updates the status, increments the winner's score, and calls `highlightWin`.
- If `moves >= 9` with no winner, it is a draw: increment `scores(draw)`.
- Otherwise it switches the player and updates the status text and color.
- `proc checkWin {}` - iterates a list of the eight winning index triples. For each, it reads the three cells; if they are all equal and non-empty, it returns the winning player. This demonstrates iterating a list of lists with nested `lindex`.
- `proc highlightWin {winner}` - finds the winning line again and sets those buttons' background to light green, then disables all remaining buttons.
- `proc newGame {}` - resets the board list, player, flags, and moves, and restores each button to blank, normal, and the default background. Note `[lindex [.board.b$i configure -bg] 3]` reads the default background from the option database so the reset restores the original color.
- `proc resetScores {}` - zeroes the scoreboard array; the labels update automatically through `-textvariable`.

> **Tip:** Storing the board as a plain list and using `lreplace` to update a
> cell is simple and clear. Because `lreplace` returns a *new* list, you must
> assign its result back to the variable (`set board [lreplace ...]`); a common
> bug is to call `lreplace` without capturing the result.

Concepts in Depth

Modeling the board as a list

The board is a list of nine strings. Reading a cell is `lindex $board $idx`; writing is `set board [lreplace $board $idx $idx $value]`. Because `lreplace`

returns a new list, you must assign the result back - forgetting to do so is the most common list-mutation bug. Modeling the grid as a flat list with index arithmetic (`row = idx / 3`, `col = idx % 3`) is simple and avoids nested lists. For larger grids, a nested list (a list of rows) is more natural, but for 3x3 the flat list is ideal.

Win detection as data

The eight winning triples are stored as a list of lists. `checkWin` iterates this list, reading the three cells of each line and testing whether they are all equal and non-empty. Treating the winning lines as data (rather than hard-coding eight `if` tests) is concise, easy to audit, and easy to extend - you could reuse the same pattern for a larger board (like Connect Four) by swapping the data. This is a small example of the data-driven style that scales to real games.

Scoreboards with -textvariable

The scoreboard labels use `-textvariable scores(X)` (and `0`, `draw`). Because the labels are bound to array elements, incrementing a score with `incr ::scores($winner)` immediately updates the display - no configure` call needed. The `::` prefix explicitly names the global scope, which is required inside procedures (where `scores` alone would refer to a local variable).
This binding pattern is the same one used throughout the widget chapters, now applied to game state.

Disabling and restoring widgets

After a move, the chosen button is `-state disabled` so it cannot be clicked again. After a win, `highlightWin` disables all remaining buttons. `newGame` restores each button with `-state normal` and resets its background to the default, read from the option database with `[lindex [.board.b$i configure -bg] 3] (the fourth element of a configure` result is the default value). This shows how the option database underpins consistent restyling.

Generating the grid with a loop

Nine buttons are created in a single `for` loop, placed with `grid` using computed row and column, and given a callback built with `[list cellClick $i]` so each button knows its own index. `grid columnconfigure`/`rowconfigure` with `-weight 1` make the 3x3 grid expand into a square. This loop-driven, data-callback pattern is the same one used by the calculator, reinforcing how idiomatic Tcl/Tk builds UIs from data and loops rather than hand-written boilerplate.

What You Learned

- A game board can be modeled as a list of fixed length, updated with `lreplace`.
- Win detection iterates a list of winning index triples.
- Scoreboards use an array bound to labels with `-textvariable`.
- Buttons generated in a loop with `[list cellClick $i]` carry their index.
- `grid columnconfigure/rowconfigure` make an expanding square grid.

Exercises

1. Add a simple computer opponent that picks a random empty cell for "O".
2. Make the winning line flash by changing the buttons' background twice with `after`.
3. Track and display the total number of games played in addition to wins and draws.

Conclusion

You have now worked through the full Tcl/Tk language and toolkit: the syntax model, variables, expressions, control flow, procedures, strings, lists, dictionaries, file I/O, regular expressions, and then the Tk GUI layer - widgets, layout, events, canvas, menus, and dialogs - culminating in four complete applications.

The thread running through everything is the twelve-rule syntax model from the Tcl manual page:

everything is a command, words are separated by whitespace, and substitution happens left to right. Master that, and the rest is learning the command catalog - which the official man pages at tcl-lang.org document in full.

Where to go next:

- Read the full Tk commands list

and explore the themed (`ttk::`) widgets for a more modern native look.

- Study the `string`,

`list`, and

`dict` man pages for

the complete set of subcommands.

- Look at the `canvas`

man page for the full drawing API, including images and rich text.

- Package your own scripts as reusable modules with `package provide` and `package require`.

Happy coding.

Sources

This book is based on the official Tcl/Tk 8.6 documentation at tcl-lang.org.

Key references cited throughout:

- Tcl syntax overview: https://www.tcl-lang.org/man/tcl8.6/TclCmd/Tcl.htm
- set / puts: https://www.tcl-lang.org/man/tcl8.6/TclCmd/set.htm
- expr: https://www.tcl-lang.org/man/tcl8.6/TclCmd/expr.htm
- proc: https://www.tcl-lang.org/man/tcl8.6/TclCmd/proc.htm
- string: https://www.tcl-lang.org/man/tcl8.6/TclCmd/string.htm
- list: https://www.tcl-lang.org/man/tcl8.6/TclCmd/list.htm
- dict: https://www.tcl-lang.org/man/tcl8.6/TclCmd/dict.htm
- file: https://www.tcl-lang.org/man/tcl8.6/TclCmd/file.htm
- open: https://www.tcl-lang.org/man/tcl8.6/TclCmd/open.htm
- regexp: https://www.tcl-lang.org/man/tcl8.6/TclCmd/regexp.htm
- Tk commands overview: https://www.tcl-lang.org/man/tcl8.6/TkCmd/contents.htm
- pack: https://www.tcl-lang.org/man/tcl8.6/TkCmd/pack.htm
- grid: https://www.tcl-lang.org/man/tcl8.6/TkCmd/grid.htm
- bind: https://www.tcl-lang.org/man/tcl8.6/TkCmd/bind.htm
- canvas: https://www.tcl-lang.org/man/tcl8.6/TkCmd/canvas.htm
- menu: https://www.tcl-lang.org/man/tcl8.6/TkCmd/menu.htm